TECHNICAL WHITE PAPER - **FEBRUARY 2019**

POWERCLI COOKBOOK FOR VMWARE VSAN

VERSION 1.4

vmware®

Table of Contents

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Introduction

Typically, vSAN management is performed through the vSphere® Client. Tasks can include those such as initial configuration, ongoing maintenance, and reporting of capacity, performance, or health of vSAN. The Web Client provides comprehensive element management of each component of a vSAN cluster.

While most element management is easily accomplished with the vSphere Client, performing many repeatable tasks across multiple clusters is largely a manual process.

While many aspects of vSAN management are automated, such as periodic health checks, error reporting, and capacity reporting, these automated tasks are specific to each individual vSAN cluster, and often have to be repeated many times when managing multiple independent vSAN clusters.

Consistency and repeatability is a challenge when performing tasks manually. It is quite common to leverage tools such as an Application Programming Interface (API) along with code to execute tasks in a consistent and repeatable fashion across one or more environments.

Microsoft® officially released PowerShell, in November of 2006, as a task automation and configuration framework. PowerShell gave administrators the ability to use a new command shell and scripting language to accomplish administrative tasks on one or more Microsoft Windows® systems more easily through the use of specialized .NET classes, called cmdlets, to perform specific operations.

PowerShell could be then be expanded through the addition of third-party modules that include one or more cmdlets and functions to accomplish additional application-centric operations. It was advantageous for vendors with Windows applications or services to provide their own PowerShell tie-ins, because administrators could accomplish both Windows and Application tasks using the same framework.

VMware PowerCLI is one such third-party add-on to Microsoft PowerShell. Virtualization administrators have long managed VMware vSphere environments, often comprised of tens, hundreds, or thousands of Microsoft Windows guests using PowerShell and PowerCLI. VMware PowerCLI over 600 cmdlets for managing and automating vSphere, vSAN, and other VMware products and solutions.

Using the PowerShell framework, along with PowerCLI, provides a robust platform to manage VMware vSphere environments at any scale.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Expectations

This document is intended to assist you with understanding types of things that can be managed programmatically through the VMware PowerCLI as they relate to vSAN.

It is neither comprehensive in showing all actions nor prescriptive in showing the only way to accomplish these tasks. This document will focus on the use of PowerCLI.

Throughout the document we will alternate showing what types of tasks can be done through the vSphere Client interface or ESXi command line, and then how to achieve the same result through PowerCLI. None of the included code samples are supported by VMware and are merely representative of ways to tasks could be accomplished.

The samples included in this document have little to no error handling. Should the foundation of these code samples be used for production code, it is recommended to include proper error handling.

Many vSAN tasks can be natively accomplished through shipping PowerCLI cmdlets. This document uses the most recent version of PowerCLI available (PowerCLI 11.2) as of this writing.

In some examples however, native cmdlets are not available to perform the required steps. For cases such as this, the vSAN Management API is directly accessed using the Get-VsanView cmdlet.

vSAN Management API

The vSAN Management API extends upon the vSphere API.

This API is exposed by both vCenter Server managing vSAN, as well as VMware ESXi hosts. Setup and all configuration of aspects of vSAN, as well as runtime state, is available by utilizing the vSAN Management API.

There are a variety of vSphere Managed Objects exposed by the vSAN Management API that provide functionality specific to vCenter Server, ESXi, or both. These Managed Objects are:

Managed Object	Function	Available
VsanVcDiskManagementSystem	vSAN Cluster configuration and query APIs for disks	vCenter
VsanVcStretchedClusterSystem	vSAN Stretched Cluster related configuration and query APIs	vCenter
VsanVcClusterConfigSystem	vSAN Cluster configuration setting and query APIs	vCenter
VsanVcClusterHealthSystem	vSAN Cluster health related configuration and query APIs	vCenter
VsanSpaceReportSystem	vSAN Cluster space usage related query APIs	vCenter
VsanPerformanceManager	vSAN Cluster performance related configuration & query APIs	vCenter & ESXi
VsanObjectSystem	vSAN Cluster setting APIs for object status query and storage policy	vCenter &

		ESXi
HostVsanSystem	Host level vSAN related configuration and query APIs	ESXi
HostVsanHealthSystem	Host level vSAN Health related configuration and query APIs	ESXi
VsanUpgradeSystem	Used to perform and monitor vSAN on-disk format upgrades. *VsanUpgradeSystemEx deprecates VsanUpgradeSystem*	vCenter
VsanUpgradeSystemEx		vCenter

Table 1 – Managed Objects presented by the vSAN API

Reliance on additional VMware APIs
It is important to also consider that vSAN is a component of vSphere. In many cases, configuration or management tasks require calls to other APIs in the VMware stack, such as the vSphere Management API.

Such tasks could include tagging a VMkernel for a specific traffic type (such as "vSAN Traffic") or configuring a host's NTP settings. These are vSphere related PowerCLI operations that could be used for environments that do not have vSAN.

Managing vSAN with PowerCLI is essentially managing the combination of vSphere and vSAN.

This document focuses primarily on using PowerCLI 11.1 with vSAN 6.7 and vSAN 6.7 Update 1. Many of the scripts could potentially work with previous versions of vSphere and vSAN, but are not guaranteed to.

Getting Started

PowerShell, or the more recent PowerShell Core are a primary requirement to be able to use PowerCLI. But which do you choose?

PowerShell is an included component of Microsoft Windows desktop and server operating systems. Different releases of Microsoft Windows have an included release of PowerShell. These may or may not be part of a default installation but can be easily added at a later time.

Native Windows applications are often tied to the specific version of Windows they are included with. Upgrading a Windows operating system to a newer release is often required to add functionality or allow compatibility with more recently released applications.

Since its initial introduction, updates to PowerShell have largely been available across different Windows operating systems. Administrators that are using older Windows operating systems, such as Windows 7 or 8 have been able to use the most recent updates to PowerShell.

Even with the ability to run newer PowerShell builds on older operating systems, there was still a requirement for Windows to be able to use PowerShell. Administrators that largely used non-Windows systems would often have to use a Windows "administrative console" or "jump-box" to use PowerShell.

In 2016, Microsoft released PowerShell Core for use on non-Windows operating systems. The release of PowerShell Core removed the requirement for a Windows operating system for many of the core capabilities of PowerShell. However, there are still some operations that still require PowerShell, because the functionality has not been added to PowerShell Core.

Tool Selection

Now that we know that PowerShell and PowerShell Core are two similar frameworks, which one is the best to use? And once we've selected one of those, what's the best coding tool to create and modify scripts?

PowerShell or PowerShell Core?

When choosing either PowerShell or PowerShell Core, it is important to consider what you want to be able to accomplish from a PowerCLI perspective.

Why is this important? Keep in mind that not all PowerShell modules have been ported to PowerShell Core. At the same time, not all PowerCLI module capabilities have been ported over either.

Each new release of PowerCLI closes the gap of which operations are available when used in conjunction with PowerShell versus PowerShell Core. With the release of PowerCLI 11, support for vCloud Director was added. Alternatively, some operations may still require PowerShell to function, such as PowerCLI ImageBuilder capabilities.

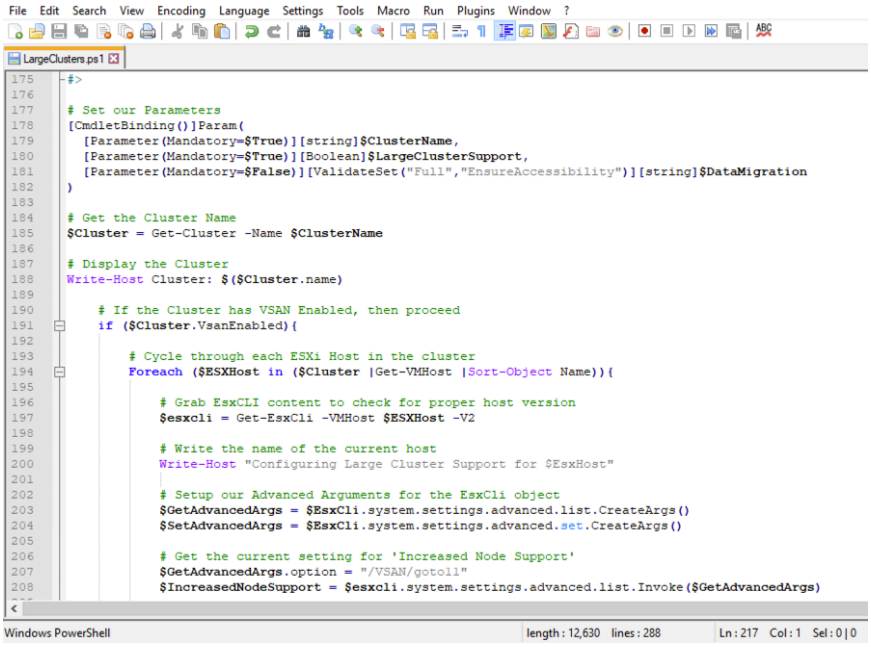
Take the types of tasks you wish to accomplish into consideration when deciding whether to use PowerShell or PowerShell Core.

Refer to the PowerCLI documentation under Automation Tools on the VMware Code site (https://code.vmware.com/tools) for the most up to date information about the requirements of PowerCLI.

Coding Tools?
PowerShell scripts typically are just text files with a .ps1 extension. They can be edited with any one of many text editors. There is no real requirement to have any particular application for creating PowerShell Scripts.

Using a text editor
Good old-fashioned Notepad in Windows is an example of a simple editor that can be used to write PowerShell scripts. If using a Mac, TextEdit is a similar offering that can suffice as well.

While these can be used accomplish the task of writing scripts, there are alternatives that may provide a more robust experience.



Notice in the above illustration that this editor (Notepad++) natively highlights syntax of the code being written.

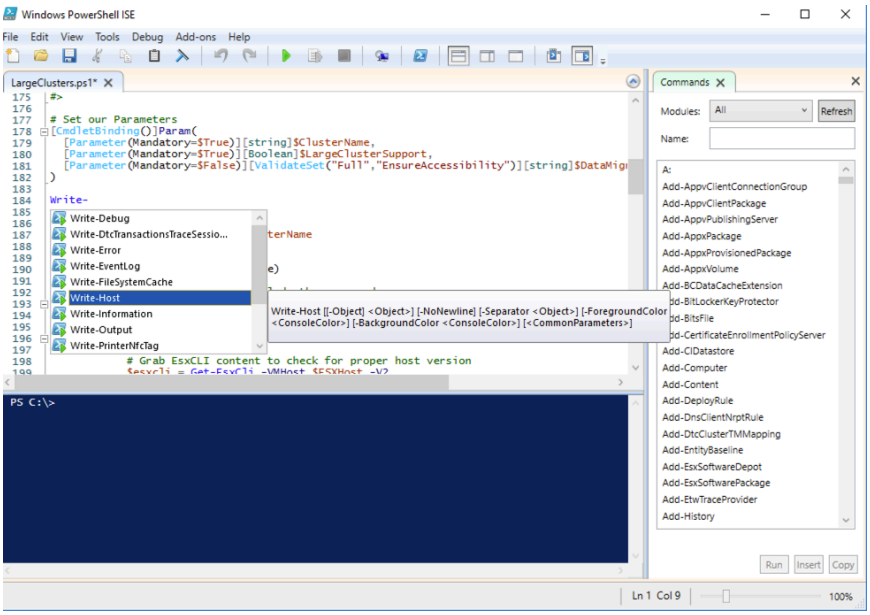
Text editors that have the ability to highlight syntax natively can make the scripting process significantly easier, especially when troubleshooting.

Using an Integrated Scripting Environment
Included with more recent releases of Windows, Microsoft included an Integrated Scripting Environment, or ISE, to help with the scripting process.

Windows PowerShell ISE goes a bit further than a simple text editor that highlights code syntax.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

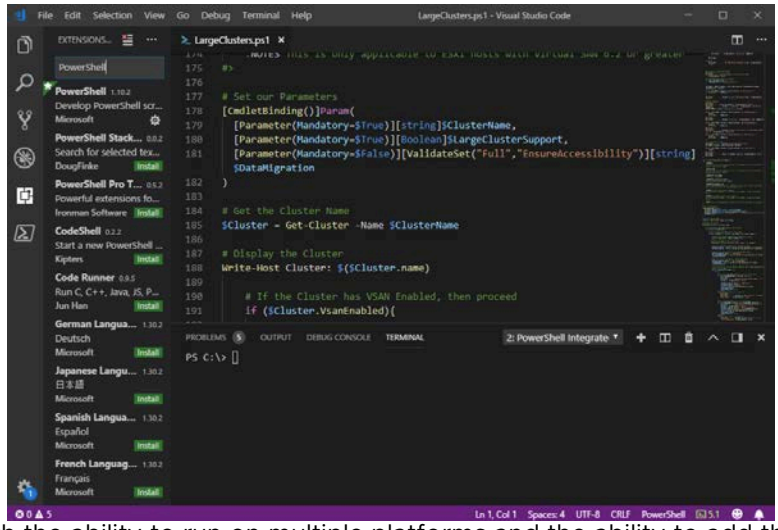
The Integrated Scripting Environment adds additional capabilities like providing “command completion”, variable property completion, “bracket matching” as code is being written, and the ability to highly code in the editor and execute it in the session below.



The Windows PowerShell ISE adds debugging and a console to actively validate the code being written from within the same interface.

With all of its integrated features, using an ISE can streamline the process of creating and testing PowerShell code. The Windows PowerShell ISE is only available on Windows platforms.

For those that wish to use an alternate ISE for environments such as Linux or Mac OS X (as well as Windows), Microsoft released Visual Studio Code. Visual Studio Code is a free open-source code editor that has many of the same features as the Windows PowerShell ISE, and more, such as adding extensions to further its capabilities. An additional benefit of Visual Studio Code is that it supports many languages other than PowerShell.



With the ability to run on multiple platforms and the ability to add third-party extensions, Visual Studio Code has largely replaced Windows PowerShell ISE.

Installing PowerShell

Microsoft's documentation is the best reference for getting started with the installation of PowerShell on various platforms.

Detailed instructions for installing Windows PowerShell or PowerShell Core can be found on the Microsoft docs site: https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-6

Installing PowerCLI

PowerCLI was originally distributed as a binary that had to be downloaded from VMware.com. Currently, PowerCLI can be installed from the PowerShell Gallery from a PowerShell session:

```
Install-Module -Name VMware.PowerCLI
```

If the account installing doesn't have administrative credentials, PowerCLI can be installed in the Scope of the Current User:

```
Install-Module -Name VMware.PowerCLI -Scope:CurrentUser
```

More detailed PowerCLI resources can be found on VMware's Code site: https://code.vmware.com/web/dp/tool/vmware-powercli/.

PowerCLI Recipes for vSAN

'Recipes' are included in this document to detail the process of how one would go about creating PowerCLI scripts for vSAN.

These will primarily be code snippets included in this document. Each recipe will include a link to a competed sample script in the respective summary section.

The majority of code listed in this document can be used on both PowerShell and PowerShell Core platforms unless otherwise indicated.

Important Note: The code samples included in this document are not supported by VMware. The code included is only provided as sample code for the purpose of demonstrating different tasks using PowerCLI.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Configuration Recipes

Configuration of vSAN is a great place to start, as all environments need to be properly configured.

A few recipes that will be covered in this section include some tasks that are vSphere related (because vSAN is part of vSphere) and some tasks that are uniquely specific to vSAN.

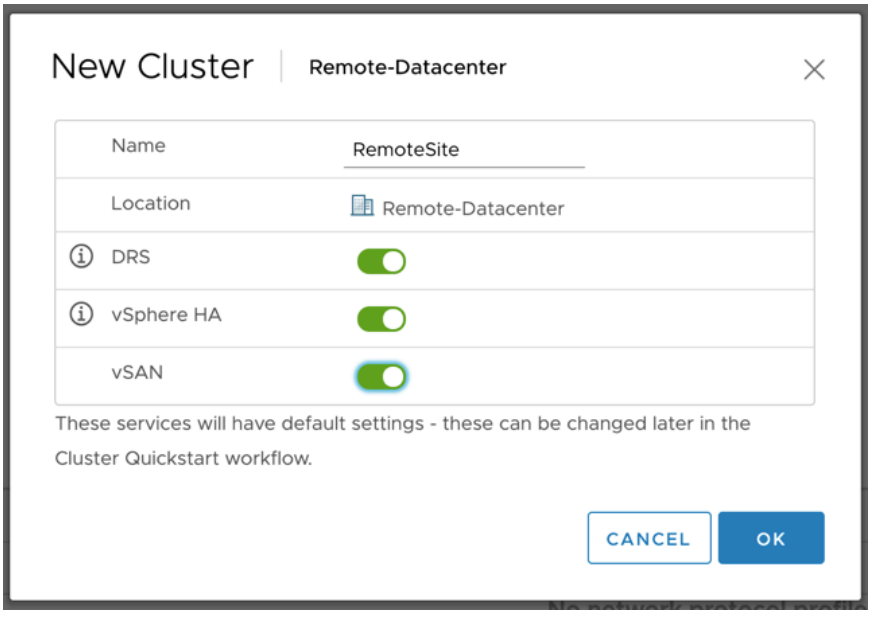
The recipes that will be covered include:

- Enabling vSAN on a new or existing Cluster
- Adding hosts to the vSAN Cluster
- Configuring vSAN Networking
- Claiming Disks for use by vSAN
- Configuring HA and DRS
- Configuring Deduplication and Compression
- Configuring vSAN Encryption
- Configuring the vSAN Performance Service

Enabling vSAN on a vSphere Cluster

For a vSphere Cluster to provide services, those services must be enabled on the vSphere Cluster. Services include vSphere Availability, vSphere Distributed Resource Scheduling, and vSAN.

Each of these services must be enabled for the Cluster to use them. In the vSphere UI this can be easily accomplished during cluster creation –



The vSphere Cluster Wizard above has a few possible inputs, which include:

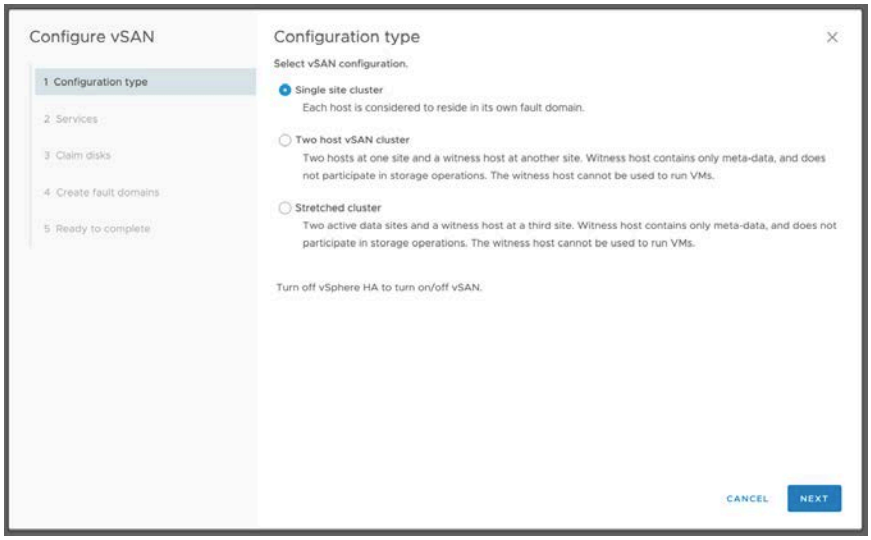
- Name
- Location (typically a Datacenter)
- vSphere DRS setting
- vSphere HA setting
- vSAN setting

These are attributes for the cluster.

Creating a cluster in PowerCLI, we must also specify these:

```
New-Cluster -Name "Cluster" -Location "Remote-Datacenter"
-HAEnabled -DrsEnabled -VsanEnabled
```

Or vSAN can be enabled after the cluster has been created:



This wizard will walk you through the process of enabling vSAN on the vSphere Cluster, as well as enable additional settings, claim disks, create fault domains, and select a vSAN Witness Host if using 2 Node or Stretched Clusters.

The vSAN Configuration Wizard is accomplishing each of these tasks through separate API calls. Using PowerCLI to do the same will take several more steps.

Enabling vSAN on an existing cluster adds the vSAN service

```
Get-Cluster -Name “vSAN” | Set-Cluster -Name "Cluster" -VsanEnabled
$true -Confirm:$false
```

Before attempting to do this in PowerCLI, let's consider what the Cluster Wizard is prompting for. The wizard has options for the cluster name, the location in the datacenter, whether vSphere HA, vSphere DRS, or vSAN are going to be enabled.

The following sample will create a new cluster in the "Remote-Datacenter" and enable HA, DRS, and vSAN.

```
New-Cluster -Name "vSAN" -Location "Remote-Datacenter" -HAEnabled -
DrsEnabled -VsanEnabled
```

The cluster is created, but none of these have been configured as of yet.

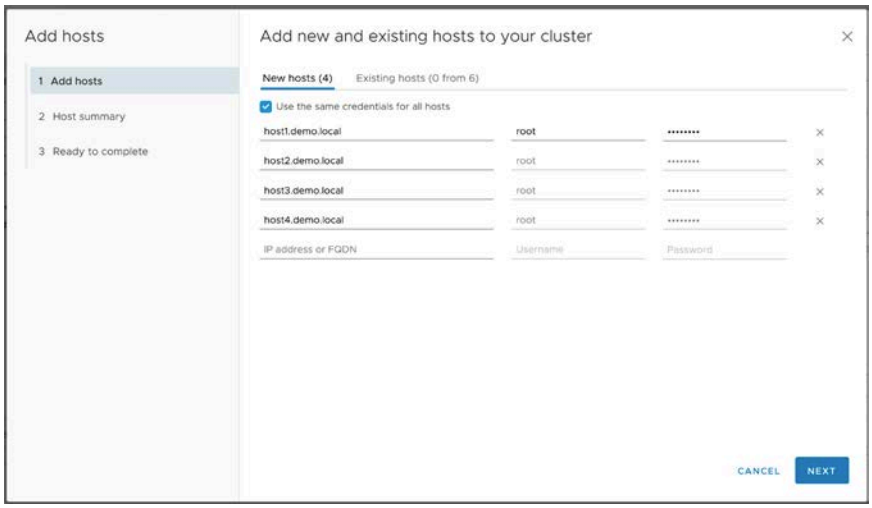
Adding hosts to a vSAN cluster

Adding hosts to a vSphere cluster has long been a manual task, accomplished serially, one host at a time in the vSphere UI.

New Hosts

The Cluster Quickstart Wizard has made this significantly easier, allowing one or more hosts to be added in a single wizard.

Hosts can be manually added:



Hosts can be added to a cluster in PowerCLI in much the same way.

Adding a single host to vCenter and a vSAN Cluster:

```
Add-VMHost -Name "HostName" -Location $Cluster -user "root" -password
"password"
```

If the host has not previously been added to vCenter, use **-Force** to accept the SSL Certificate to proceed

```
Add-VMHost “hostname” -Location $Cluster -user “root” -password
“password” -Force
```

Multiple hosts not part of vCenter could be added from an array:

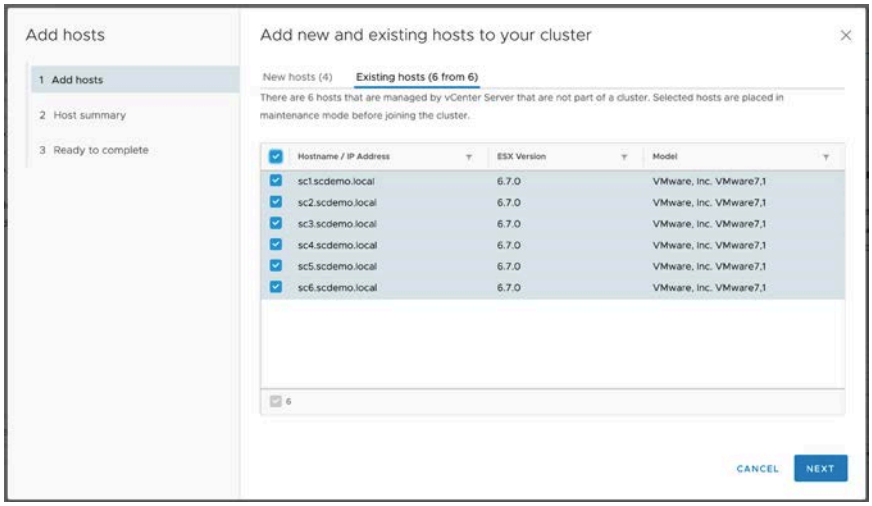
```
$HostList = (“host1”,”host2”,”host3”,”host4”)

Foreach ($Item in $HostList) {
   Add-VMHost $Item -Location $Cluster -user “root” -password
“password” -Force
}
```

If it isn’t desired to put credentials in the script, they can be prompted for, or possibly read from an external file that has permissions secured for only authorized administrators:

```
$HostCreds = Get-VICredentialStoreItem -File “C:\Secure\Creds.xml”

$HostList = (“host1”,”host2”,”host3”,”host4”)

Foreach ($Item in $HostList) {
   Add-VMHost $Item -Location $Cluster -Credentials $HostCreds -Force
}
```

Existing Hosts

In the new Cluster Quickstart, hosts can also be added if they are already present in vCenter:



This is a bit more difficult though, because Get-VMHost simply returns all hosts attached to vCenter:

```
PS /> Get-VMHost

Name                 ConnectionState PowerState
----                 --------------- ----------
sc6.scdemo.local     Connected       PoweredOn
sc3.scdemo.local     Connected       PoweredOn
sc5.scdemo.local     Connected       PoweredOn
sc2.scdemo.local     Connected       PoweredOn
sc1.scdemo.local     Connected       PoweredOn
sc4.scdemo.local     Connected       PoweredOn
witness1.scdemo.l... Connected       PoweredOn
```

What if we only want to get a list of hosts that are in a specific datacenter?

```
PS /> $Datacenter = Get-Datacenter -Name “Datacenter”
PS /> Get-VMHost -Location $Datacenter

Name                 ConnectionState PowerState
----                 --------------- ----------
sc6.scdemo.local     Connected       PoweredOn
sc3.scdemo.local     Connected       PoweredOn
sc5.scdemo.local     Connected       PoweredOn
sc2.scdemo.local     Connected       PoweredOn
sc1.scdemo.local     Connected       PoweredOn
sc4.scdemo.local     Connected       PoweredOn
```

This still doesn’t differentiate between hosts that are in a cluster or not. Looking at the full properties of Get-VMHost, the Parent value will indicate whether the host is a member a cluster or not.

```
PS /> Get-VMHost -Location $Datacenter | Where-Object {$_.Parent.Name
-eq “host”}

Name                 ConnectionState PowerState
----                 --------------- ----------
sc6.scdemo.local     Connected       PoweredOn
sc3.scdemo.local     Connected       PoweredOn
sc5.scdemo.local     Connected       PoweredOn
sc2.scdemo.local     Connected       PoweredOn
sc1.scdemo.local     Connected       PoweredOn
sc4.scdemo.local     Connected       PoweredOn
```

The VMHost Parent.Name value will be “host” for hosts that are not part of a vSphere Cluster. Using that logic, it should be easy to add all available hosts to the vSAN Cluster.

```
PS /> Get-VMHost -Location $Datacenter

Name                 ConnectionState PowerState
----                 --------------- ----------
sc6.scdemo.local     Connected       PoweredOn
sc3.scdemo.local     Connected       PoweredOn
sc5.scdemo.local     Connected       PoweredOn
```

```
sc2.scdemo.local     Connected       PoweredOn
sc1.scdemo.local     Connected       PoweredOn
sc4.scdemo.local     Connected       PoweredOn
```

For hosts that are added to vCenter, but not part of a cluster, the Add-VMHost cmdlet isn't used. Instead, the Move-VMHost cmdlet is used to move them into a cluster.

```
PS /> Move-VMHost -VMHost "sc1.scdemo.local" -Location $Cluster

Name                 ConnectionState PowerState
----                 --------------- ----------
sc1.scdemo.local        Connected    PoweredOn
```

By combining the code to list all nodes not in a cluster with the Move-VMHost cmdlet, it is easy to add each of these hosts.

```
PS /> Get-VMHost -Location $Datacenter| Where-Object {$_.Parent.Name -
eq "host"} | Move-VMHost -Location $Cluster

Name                 ConnectionState PowerState
----                 --------------- ----------
sc1.scdemo.local        Connected    PoweredOn
sc6.scdemo.local        Connected    PoweredOn
sc3.scdemo.local        Connected    PoweredOn
sc5.scdemo.local        Connected    PoweredOn
sc2.scdemo.local        Connected    PoweredOn
sc4.scdemo.local        Connected    PoweredOn
```

Converting a Cluster to a Stretched Cluster

Another common vSAN Cluster task, is configuring the cluster as a Stretched Cluster.

vSAN Stretched Clusters can have between 1 and 15 hosts per site (typically different locations) and require a vSAN Witness Host in a third location. Hosts in each site are assigned in to fault domain, with one of those fault domains being designated as the Preferred site.

2 Node vSAN is architecturally the same as a Stretched Cluster configuration with a single host in each fault domain, and both hosts residing in the same physical location.

This section will only cover the mechanics of creating fault domains, assigning hosts to those fault domains, and setting the cluster to a stretched configuration. Networking requirements will be covered in the Configuring vSAN Networking section.

The New-VsanFaultDomain cmdlet is used to create a fault domain. Requirements of this cmdlet include the name of the new fault domain, and the hosts that will reside in the fault domain.

```
PS /> New-VsanFaultDomain -Name "Preferred" -VMHost
"sc1.scdemo.local","sc2.scdemo.local”,"sc3.scdemo.local"
```

```
Name                           Cluster
----                           -------
Preferred                      vSAN

PS /> New-VsanFaultDomain -Name "NonPreferred" -VMHost
"sc4.scdemo.local","sc5.scdemo.local","sc6.scdemo.local"

Name                           Cluster
----                           -------
NonPreferred                   vSAN
```

Manually having to add each of hosts into the New-VsanFaultDomain cmdlet can be a bit of time consuming. A good alternative to using the actual ESXi host names, is to add those host names to an array and running the New-VsanFaultDomain cmdlet.

```
PS /> $Primary = “sc1.scdemo.local”,”sc2.scdemo.local”,”sc3.scdemo.local”
PS /> New-VsanFaultDomain -Name "Primary" -VMHost $Primary
```

Going a bit further, it is possible to dynamically put hosts into arrays. It is common for Stretched Cluster configurations to have a uniform (even) number, so why not simply split the list of hosts down the middle?

```
# Put the hosts into an array, sorted by name
$ESXhosts = Get-Cluster -Name vSAN | Get-VMHost | Sort-Object Name

# The middle of the array is the count divided by 2
# An odd count will make the first array larger by 1
$ArrayMiddle = [int](($ESXhosts.Count)/2)

# Put the first half of hosts in Primary & second half in Secondary
$Primary = $ESXhosts[0..(($ArrayMiddle)-1)]
$Secondary = $ESXhosts[$ArrayMiddle..(($ESXhosts.Count)-1)]

# Create the fault domains & store the objects in variables to use later
$PrimaryFD = New-VsanFaultDomain -Name "Primary" -VMHost $Primary
$SecondaryFD = New-VsanFaultDomain -Name "Secondary" -VMHost $Secondary
```

Invoking the variable names, it can be seen that each has 3 hosts of the 6 host vSAN cluster.

```
PS /> $Primary

Name                 ConnectionState PowerState NumCpu Version
----                 --------------- ---------- ------ -------
sc1.scdemo.local     Connected       PoweredOn       2   6.7.0
sc2.scdemo.local     Connected       PoweredOn       2   6.7.0
sc3.scdemo.local     Connected       PoweredOn       2   6.7.0


PS /> $Secondary

Name                 ConnectionState PowerState NumCpu Version
----                 --------------- ---------- ------ -------
sc4.scdemo.local     Connected       PoweredOn       2   6.7.0
sc5.scdemo.local     Connected       PoweredOn       2   6.7.0
sc6.scdemo.local     Connected       PoweredOn       2   6.7.0
```

The Get-VsanFaultDomain cmdlet will list the fault domains:

```
PS /> Get-VsanFaultDomain

Name                           Cluster
----                           -------
Primary                        vSAN
Secondary                      vSAN
```

With the fault domains configured, the next step is to change the cluster's configuration to Stretched. One of the fault domains will have to be set as the Preferred, and a vSAN Witness Host will have to be designated.

The process of designating the vSAN Witness Host simply requires the Witness Host name, but it is important to know which devices will be used for the Witness disk group cache and capacity devices.

For customers using the vSAN Witness Appliance with the Tiny or Normal profile, the cache device typically has the device id of mpx.vmhba1:C0:T2:L0 and the capacity device has the device id of mpx.vmhba1:C0:T1:L0.

Assuming a vSAN Witness Host has been deployed, the Set-VsanClusterConfiguration cmdlet is used to convert an existing cluster to a stretched cluster.

```
PS /> $Cluster = Get-Cluster -Name "vSAN"
PS /> Set-VsanClusterConfiguration -Configuration $Cluster -StretchedClusterEnabled
$True -PreferredFaultDomain $PrimaryFD -WitnessHost witness1.scdemo.local -
WitnessHostCacheDisk mpx.vmhba1:C0:T2:L0 -WitnessHostCapacityDisk mpx.vmhba1:C0:T1:L0

Cluster              VsanEnabled  IsStretchedCluster   Last HCL Updated
-------              -----------  ------------------   ----------------
vSAN                 True         True                 1/23/19 11:45:00 AM
```

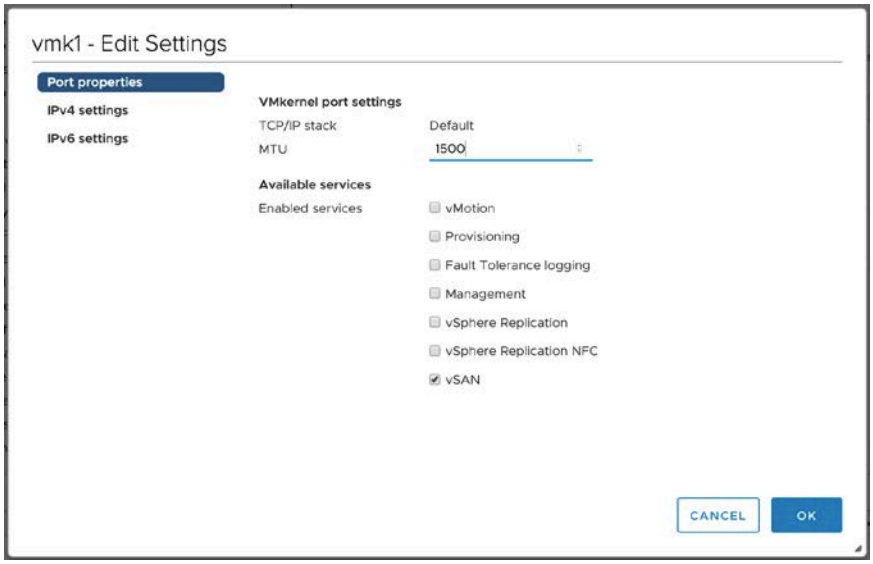
Configuring vSAN Networking

A VMkernel interface tagged for vSAN Traffic is required for vSAN nodes to communicate with each other. This is true for vSAN nodes that are presenting vSAN disk groups, or for vSAN nodes that are only consuming vSAN storage.

The VMkernel interface used could be the default configured Management VMkernel interface (vmk0) or a dedicated VMkernel interface. It is a more common practice to isolate storage traffic to an alternate VMkernel interface, so we will proceed with the assumption and expectation that we're using an alternate interface.

Tagging an existing VMkernel adapter

If an existing VMkernel adapter is already present, vSAN traffic only needs to be tagged. The process is the same, whether the existing VMkernel adapter is attached to a vSphere Standard Switch or vSphere Distributed Switch.



The Set-VMHostNetwork Adapter has vSAN Traffic as an argument and allows for a simple "one-liner":

```
PS /> Get-VMHostNetworkAdapter -Name "vmk1" -VMHost "sc1.scdemo.local" |  Set-
VMHostNetworkAdapter -VsanTrafficEnabled $true
Perform operation?
Performing operation 'Configuring VM host network adapter.' on network adapter with IP
'192.168.110.31' and device name 'vmk1'.
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Yes"): A

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
----       ---               ----------- --              ----------      ----------
vmk1       00:50:56:67:4e:48 False       192.168.110.31  255.255.255.0         vmk1
```

To perform this without being prompted to confirm, simply append -Confirm:$false to the end of the one-liner.

```
PS /> Get-VMHostNetworkAdapter -Name "vmk1" -VMHost "sc1.scdemo.local" |  Set-
VMHostNetworkAdapter -VsanTrafficEnabled $true -Confirm:$false

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
----       ---               ----------- --              ----------      ----------
vmk1       00:50:56:67:4e:48 False       192.168.110.31  255.255.255.0         vmk1
```

It is also important to *specify the host* that this action is performed on. Omitting the VMHost property will perform this across all hosts connected to vCenter:

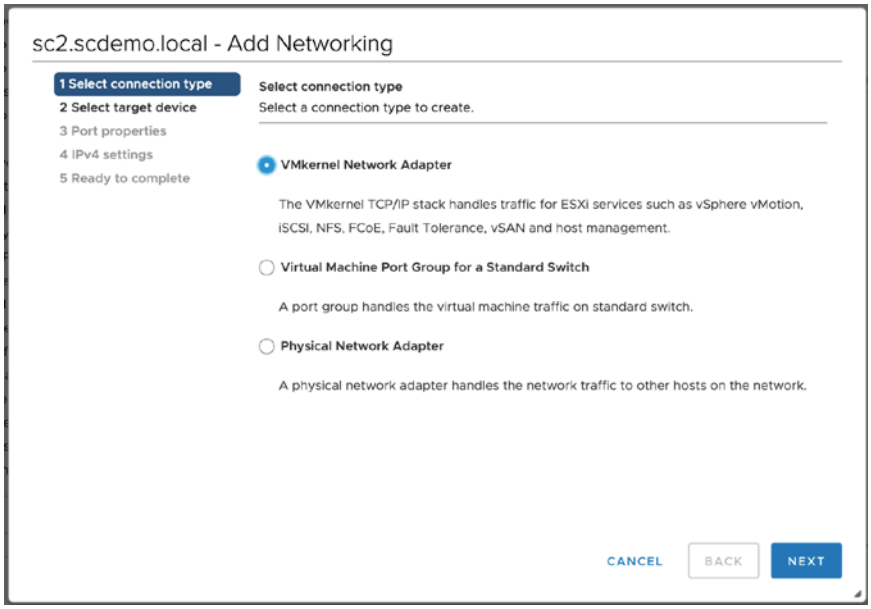
```
PS /> Get-VMHostNetworkAdapter -Name "vmk1" | Set-VMHostNetworkAdapter -
VsanTrafficEnabled $true
Perform operation?
Performing operation 'Configuring VM host network adapter.' on network adapter with IP
'192.168.110.31' and device name 'vmk1'.
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Yes"): A

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
```

```
----       ---               ----------- --              ----------      ----------
vmk1       00:50:56:67:4e:48 False       192.168.110.31  255.255.255.0         vmk1
vmk1       00:50:56:69:bc:87 False       192.168.110.32  255.255.255.0         vmk1
vmk1       00:50:56:82:d6:a2 False       192.168.137.201 255.255.255.0         vmk1
vmk1       00:50:56:82:b6:6c False       192.168.137.202 255.255.255.0         vmk1
```

This is likely not a desirable, as it could impact services on hosts that do not require vSAN tagging on vmk1.

Creating a new VMkernel Adapter on a vSphere Standard Switch

What if a VMkernel adapter isn't already configured on a host? The vSphere Client provides a wizard to create a new VMkernel adapter, configure the vSwitch that it is attached to, as well as services tagging, IP addressing, and more.



This is relatively easy as well with PowerCLI. The New-VMHostNetworkAdapter cmdlet will create the VMkernel adapter.

```
PS /> New-VMHostNetworkAdapter -VMHost "hostname" -PortGroup "vSAN" -VirtualSwitch
"vSwitch0" -IP 192.168.110.33 -SubnetMask 255.255.255.0 -Mtu 1500 -VsanTrafficEnabled
$true

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
----       ---               ----------- --              ----------      ----------
vmk1       00:50:56:67:76:f7 False       192.168.110.33  255.255.255.0         vmk1
```

The New-VMHostNetworkAdapter does not provide a mechanism to set the VLAN to be used by the VMkernel interface. If a specific VLAN is to be used by the newly created VMkernel interface, the Set-VirtualPortGroup cmdlet will need to be used:

```
PS /> Get-VirtualPortGroup -VMHost “hostname" -Name "vSAN" | Set-VirtualPortGroup -
VLanId 100

Name                    Key                           VLanId PortBinding NumPorts
----                    ---                           ------ ----------- --------
vSAN                    key-vim.host.PortGroup-vSAN   100
```

All default vSphere installations include a the vSwitch0 vSphere Standard switch upon the initial installation.

Using a vSphere Distributed Switch for vSAN

vSAN licensing unlocks the vSphere Distributed Switch functionality for vSphere environments that are not licensed with vSphere Enterprise Plus licensing. *In this situation, use of a vSphere Distributed Switch would require the cluster to be licensed for vSAN and a vSAN datastore configured.*

Using a vSphere Distributed Switch vSAN not only provides additional features, but also the ability to manage network settings for different port groups uniformly across all hosts. This is especially helpful when making consistent configuration changes for a vSAN cluster.

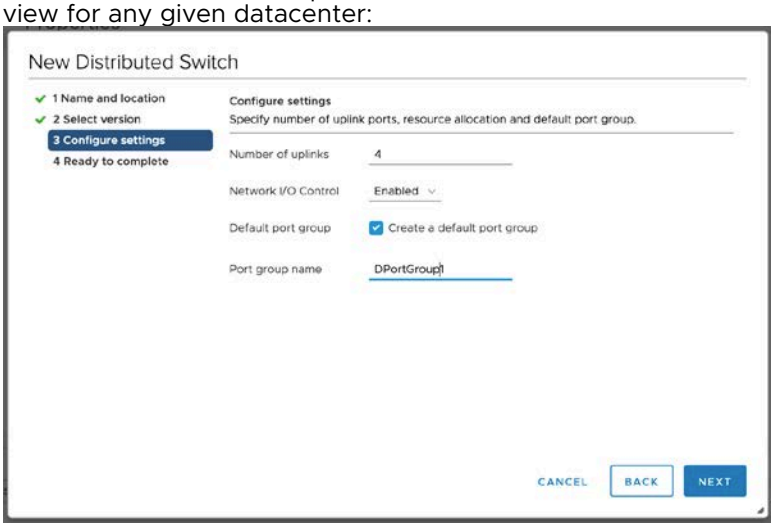
In the previous example, a vSphere Standard Switch was used. Every vSphere host has a vSwitch configured by default.

vSphere Distributed Switches have to be created for each vSphere Datacenter, and hosts must be attached to the VDS individually.

Only after the VDS has been created and hosts have been attached (with some physical uplinks) can a VDS be used for a vSAN Cluster.

Creating a vSphere Distributed Switch

A VDS must be created in a vSphere Datacenter to be used by hosts in that datacenter. In the vSphere Client this is done from the Networking view for any given datacenter:



Different settings that may be set with the wizard include the name, version, number of uplinks from each host, whether Network IO Control is enabled or not, and whether or not to create a default port group.

If you aren't familiar with Network IO Control (NIOC), it is a feature that will let an administrator assign shares, reservations, and limits to different traffic types. More specific details for NIOC in vSphere 6.7 can be found here.

Creating a VDS is performed with the New-VDSwitch cmdlet. Remember that a VDS is located in a Datacenter, so the Datacenter must be specified when creating the VDS.

```
PS /> New-VDSwitch -Name "VDSwitch" -Location (Get-Datacenter -Name "DC")

Name                           NumPorts   Mtu        Version  Vendor
----                           --------   ---        -------  ------
VDSwitch                       0          1500       6.6.0    VMware, Inc.
```

To mimic the behavior of the Wizard, some additional parameters will need to be set, like the number of uplinks and Maximum Transmission Unit (MTU) size. NIOC requires some additional steps, which will be covered shortly.

```
PS /> New-VDSwitch -Name "VDSwitch" -Location (Get-Datacenter -Name "DC") -Mtu 9000 -
NumUplinkPorts 4 -Version "6.5.0"

Name                           NumPorts   Mtu        Version  Vendor
----                           --------   ---        -------  ------
VDSwitch                       0          9000       6.5.0    VMware, Inc.
```

Enabling NIOC isn't directly exposed with PowerCLI, so the Get-View cmdlet will be used to enable NIOC: To enable NIOC,

```
PS /> $VDSwitchView = Get-View -Id (Get-VDSwitch -Name "VDSwitch")
PS /> $VDSwitchView.EnableNetworkResourceManagement($true)
PS />
```

There are some additional settings that can be added, and can be seen by running `Get-Help New-VDSwitch -Full`.

After a VDS has been created, hosts must be added to the VDS. Adding a host to the VDS is necessary.

```
PS /> Get-VDSwitch -VMHost "sc1.scdemo.local" | Where-Object {$_.Name -eq "VDSwitch"}
PS />
PS /> If (-NOT (Get-VDSwitch -VMHost "sc1.scdemo.local" | Where-Object {$_.Name -eq
"VDSwitch"})) { Get-VDSwitch -Name "VDSwitch" | Add-VDSwitchVMHost "sc1.scdemo.local"
}
```

Once a host has been added, one or more physical NICs from the host must be added to the VDS.

```
PS /> Get-VDSwitch -VMHost "sc1.scdemo.local" | Where-Object {$_.Name -eq "VDSwitch"}
| Add-VDSwitchPhysicalNetworkAdapter -VMHostPhysicalNic (Get-VMHostNetworkAdapter -
Name "vmnic1" -VMHost "sc1.scdemo.local")
```

```
Confirm
Are you sure you want to perform this action?
Performing the operation "Adding physical network adapter(s) 'vmnic1'" on target
"VDSwitch".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Yes"): Y
PS />
```

To perform this without being prompted to confirm, simply append -Confirm:$false to the end.

```
PS /> Get-VDSwitch -VMHost "sc1.scdemo.local" | Where-Object {$_.Name -eq "VDSwitch"}
| Add-VDSwitchPhysicalNetworkAdapter -VMHostPhysicalNic (Get-VMHostNetworkAdapter -
Name "vmnic1" -VMHost "sc1.scdemo.local") -Confirm:$false
PS />
```

Those are quite a few steps. Create a VDS, add each host to the VDS, and uplink physical NICs from each of the hosts.

Here is an example of putting that all together for a new vSAN cluster when a VDS does not already exist:

```
# Get the Datacenter Object
$Datacenter = Get-Datacenter -Name "Datacenter"

# Create the new VDS named VDSwitch
$VDSwitch = New-VDSwitch -Name "VDSwitch" -Location $Datacenter -Version "6.6.0" -Mtu
1500 -NumUplinkPorts 2

# Use Get-View to set NIOC
$VDSwitchView = Get-View -Id $VDSwitch.Id
$VDSwitchView.EnableNetworkResourceManagement($true)

# Get the vSAN Cluster
$Cluster = Get-Cluster -Name "vSAN"

# Enumerate all the hosts and cycle through them
Foreach ($ESXhost in ($Cluster | Get-VMHost)) {

  # If the VDS doesn't exist on the host, add it
  If (-Not (Get-VDSwitch -VMHost $ESXhost | Where-Object {$_.Name -eq "VDSwitch1"})) {
        # Add the host to the VDS
        $VDSwitch | Add-VDSwitchVMHost -VMHost $ESXhost
        # Add pnics 2 & 3 to the VDS
        $VDSwitch | Add-VDSwitchPhysicalNetworkAdapter -VMHostPhysicalNic (Get-
VMHostNetworkAdapter -Name "vmnic2" -VMHost $ESXhost) -Confirm:$false
        $VDSwitch | Add-VDSwitchPhysicalNetworkAdapter -VMHostPhysicalNic (Get-
VMHostNetworkAdapter -Name "vmnic3" -VMHost $ESXhost) -Confirm:$false
    }
}

Name                           NumPorts   Mtu        Version  Vendor
----                           --------   ---        -------  ------
VDSwitch1                      0          1500       6.6.0    VMware, Inc.
```

The physical NIC selection could get more specific than simple vmnicX names by searching for additional properties of the NICs.

Below is sample code to query the physical NICs that have a speed greater than 1GB, and then loop through them to add them to the VDS.

```
# Grab the pNICs with >1Gbps, we'll expect any NICs with >1Gbps to be direct connected
```

```
$pNICs = $ESXHost | Get-VMHostNetworkAdapter | Where-Object {$_.BitRatePerSec -gt
"1000"}

ForEach ($pNIC in $pNICs ) {
    $VDSwitch | Add-VDSwitchPhysicalNetworkAdapter -VMHostPhysicalNic $pNIC -
Confirm:$false
}
```

Once a host or hosts are added to the VDS, a VMkernel interface for vSAN can be created.

Creating a VDS Portgroup and VMkernel Adapter for vSAN on a VDS
The cmdlet used to create a VMkernel interface on a vSphere Distributed Switch is a little different than the one used for a vSphere Standard Switch.

A portgroup must exist on the vSphere Distributed Switch, the vSAN Host must be attached to the vSphere Distributed Switch, and only then can a VMkernel interface be created for vSAN.

The New-VDPortGroup cmdlet will create the VDS portgroup.

```
PS /> New-VDPortGroup -Name “vSAN” -VDSwitch “VDSwitch” -Numports 8 -VlanID 100

Name                    Key                           VLanId PortBinding NumPorts
----                    ---                           ------ ----------- --------
vSAN                    key-vim.host.PortGroup-vSAN   100
```

The VDS portgroup only needs to be created once for the vSAN cluster, as it is attached to the entire VDS.

With the vSAN portgroup has been created, and one or more hosts have been added to the VDS, each with one or more physical adapters attached to the VDS, a new VMkernel interface can be created for use by vSAN using New-VMHostNetworkAdapter.

```
PS /> New-VMHostNetworkAdapter -VMHost "sc1.scdemo.local" -PortGroup "vSAN" -
VirtualSwitch "VDSwitch" -IP 192.168.110.31 -SubnetMask 255.255.255.0 -Mtu 1500 -
VsanTrafficEnabled $true

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
----       ---               ----------- --              ----------      ----------
vmk1       00:50:56:63:0c:1c False       192.168.110.31  255.255.255.0         vmk1


PS />
```

Unless using DHCP, a specific IP address must be assigned to the VMkernel interface. This can be quite cumbersome for configuring IP addresses for any number of nodes, large or small.

A fairly common practice for virtualization administrators, is to use matching 4th octet addresses for different interfaces on a vSphere host.

A good example of this would be something like Host 1’s configuration below:

- vmk0 – Management – 10.192.102.237
- vmk1 – vSAN – 192.168.101.237
- vmk2 – vMotion – 192.168.102.237
-

The ability to programmatically use the same 4th octet from the Management VMkernel interface is easy with the Split operation.

Before we can split though, we need to retrieve the IP address of vmk0 above. Get-VMHostNetwork will help with that.

```
PS /> $ESXhost | Get-VMHostNetwork

HostName     DomainName   DnsFro ConsoleGateway  ConsoleGatewayD DnsAddress
                          mDhcp                  evice
--------     ----------   ------ --------------  --------------- ----------
sc2-rdops... eng.vmwar... True                                   {10.195.12.31, 10...
```

Unfortunately, we need a little more information. Selecting only the VMkernelGateway, and expanding it to the virtual NIC will give us what is necessary.

```
PS /> $ESXhost | Get-VMHostNetwork | Select-Object HostName, VMKernelGateway -
ExpandProperty VirtualNic

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
----       ---               ----------- --              ----------      ----------
vmk0       02:00:2e:04:fb:b0 True        10.192.120.68   255.255.224.0         vmk0
```

In the example above, the IP is 10.192.120.68. If we put the results in a variable, the IP can be easily retrieved.

```
PS /> $vmk0 = $ESXhost | Get-VMHostNetwork | Select-Object HostName, VMKernelGateway -
ExpandProperty VirtualNic
PS /> $vmk0.IP
10.192.120.68
```

Now that the IP address of vmk0 is available as $vmk0.IP, it can be split to determine the last octet value.

```
PS /> $LastOctet = $vmk0.IP.Split('.')[-1]
```

With the ability to enumerate all the hosts in a vSAN cluster and retrieve the last octet of vmk0, it is easy to add a VDS VMkernel Interface for vSAN use across the entire cluster.

```
# Get the VDS Object
$VDSwitch = Get-VDSwitch -Name "VDSwitch"

# Get the Port Group Object
$VDPortGroup = Get-VirtualPortGroup -Name "vSAN"

# Get the Cluster object
$Cluster = Get-Cluster -Name "vSAN1"

# Set the value for the 1st 3 octets of the vSAN Network
$VsanPrefix = "192.168.101."

# Enumerate each of the hosts, retrieve the IP of vmk0, and use the last octet to
```

```
# create a vSAN VMkernel interface
Foreach ($ESXhost in ($Cluster | Get-VMHost)) {
    # Get the IP of vmk0
    $vmk0 = $ESXhost | Get-VMHostNetwork | Select-Object HostName, VMKernelGateway -
ExpandProperty VirtualNic

    # Get the last octet of vmk0
    $LastOctet = $vmk0.IP.Split('.')[-1]

    # set the full IP of the vSAN interface for the current host
    $VsanIP = $VsanPrefix+$LastOctet

    # Create the VMkernel interface
    New-VMHostNetworkAdapter -VirtualSwitch $VDSwitch -VMHost $ESXhost -PortGroup
$VDPortGroup -IP $VsanIP -SubnetMask "255.255.255.0" -VsanTrafficEnabled $true
}


The resulting output is:
192.168.101.57

Name       Mac               DhcpEnabled IP              SubnetMask      DeviceName
----       ---               ----------- --              ----------      ----------
vmk1       00:50:56:67:e1:82 False       192.168.101.57  255.255.255.0         vmk1
192.168.101.215
vmk1       00:50:56:6e:a2:fe False       192.168.101.215 255.255.255.0         vmk1
192.168.101.23
vmk1       00:50:56:66:85:7d False       192.168.101.23  255.255.255.0         vmk1
192.168.101.46
vmk1       00:50:56:61:c5:04 False       192.168.101.46  255.255.255.0         vmk1
192.168.101.237
vmk1       00:50:56:63:d9:73 False       192.168.101.237 255.255.255.0         vmk1
192.168.101.68
vmk1       00:50:56:69:00:9c False       192.168.101.68  255.255.255.0         vmk1
```

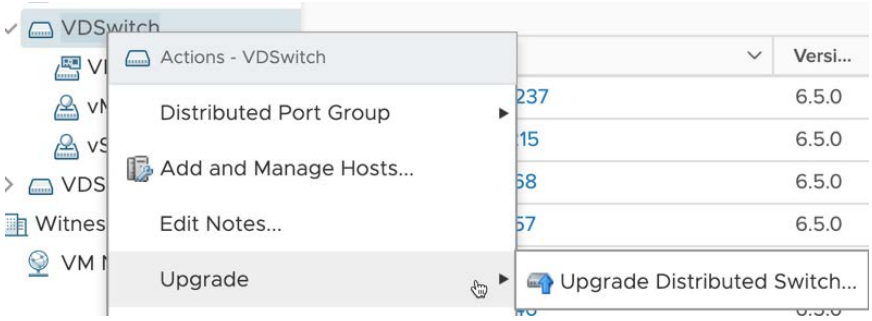
The same could be done for creating a vMotion VMkernel interface. The only code difference would be to use -VMotionEnabled instead of -VsanTrafficEnabled.

Upgrading a vSphere Distributed Switch and enabling NIOC

It has been stated a few times that vSAN is part of vSphere. Updating vSphere hosts will also update vSAN builds. Something that often goes overlooked, is upgrading the vSphere Distributed Switch version.

Consider the scenario where a customer installs vSphere 6.0 and vSAN 6.2 (the most recent release of vSAN that is part of vSphere). When the customer upgrades to vSphere 6.5 or 6.7, it is entirely possible to not upgrade the vSphere Distributed Switch.

An administrator would need to know to go into the vSphere Client, right click on the VDS and select Upgrade.



Upgrading the VDS is missed quite often. While this is easily accomplished in the UI individually, it is more problematic at scale.

Let's start with upgrading a single VDS to the latest version. There is no native PowerCLI cmdlet to upgrade a VDS, so Get-View will have to be used.

```
# Get the VDSwitch
$VDSwitch = Get-VDSwitch -Name "DSwitch0"

# Create a VDS Product Specification
$DVSProdSpec = New-Object VMware.Vim.DistributedVirtualSwitchProductSpec
$DVSProdSpec.ForwardingClass = 'cswitch'
$DVSProdSpec.Vendor = 'VMware, Inc.'
$DVSProdSpec.Name = 'DVS'
$DVSProdSpec.Version = '6.6.0'

# Apply the DVS Prod Spec with an Upgrade Operation
$VDSwitchView.PerformDvsProductSpecOperation_Task('upgrade',$DVSProdSpec)
```

Enabling NIOC is also not natively available in PowerCLI, so Get-View will have to be used to enable it as well:

```
# Get the VDSwitch
$VDSwitch = Get-VDSwitch -Name "DSwitch0"

# Enable NIOC
$VDSwitchView = Get-View -ID $VDSwitch.Id

# Change NIOC to enabled
$VDSwitchView.EnableNetworkResourceManagement($true)
```

These could be easily combined:

```
# Get the VDSwitch
$VDSwitch = Get-VDSwitch -Name "DSwitch0"

# Create a VDS Product Specification
$DVSProdSpec = New-Object VMware.Vim.DistributedVirtualSwitchProductSpec
$DVSProdSpec.ForwardingClass = 'cswitch'
$DVSProdSpec.Vendor = 'VMware, Inc.'
$DVSProdSpec.Name = 'DVS'
$DVSProdSpec.Version = '6.6.0'

# Apply the DVS Prod Spec with an Upgrade Operation
$VDSwitchView.PerformDvsProductSpecOperation_Task('upgrade',$DVSProdSpec)

# Enable NIOC
```

```
$VDSwitchView = Get-View -ID $VDSwitch.Id

# Change NIOC to enabled
$VDSwitchView.EnableNetworkResourceManagement($true)
```

With NIOC enabled, different traffic types like vSAN, vMotion, Management, VM, or other could be adjusted as well.

Looking at System Traffic in a VDS's configuration will show the different traffic types:

Traffic Type	Shares	Shares Value	Reservation	Limit
Management Traffic	Normal	50	0 Mbit/s	Unlimited
Fault Tolerance (FT) Traffic	Normal	50	0 Mbit/s	Unlimited
vMotion Traffic	Normal	50	0 Mbit/s	Unlimited
Virtual Machine Traffic	High	100	0 Mbit/s	Unlimited
iSCSI Traffic	Normal	50	0 Mbit/s	Unlimited
NFS Traffic	Normal	50	0 Mbit/s	Unlimited
vSphere Replication (VR) Traffic	Normal	50	0 Mbit/s	Unlimited
vSAN Traffic	Normal	50	0 Mbit/s	Unlimited
vSphere Data Protection Backup Traffic	Normal	50	0 Mbit/s	Unlimited

These can be enumerated using from the VDS Object:

```
PS /> $VDSwitch = Get-VDSwitch "VDSwitch"

PS /> $VDS.ExtensionData.Config.InfrastructureTrafficResourceConfig | Select-Object
Key,Description

Key            Description
---            -----------
management     Management Traffic Type
faultTolerance Fault Tolerance (FT) Traffic Type
vmotion        vMotion Traffic Type
virtualMachine Virtual Machine Traffic Type
iSCSI          iSCSI Traffic Type
nfs            NFS Traffic Type
hbr            vSphere Replication (VR) Traffic Type
vsan           vSAN Traffic Type
vdp            vSphere Data Protection Backup Traffic Type
```

The Infrastructure Traffic Resource Configuration, part of the VDS Config contains each of these.

To adjust the shares, share value, reservation, or limit of each of these traffic types, a VDS Reconfiguration Task must be accomplished.

To reconfigure a VDS, a DVSConfigSpec must be created. Each of the traffic types are an Infrastructure Traffic Resource, which must be created also. The Infrastructure Traffic Recourse is applied to the DVSConfigSpec, which is then used to reconfigure the VDS.

```
# Get the VDS object
$VDS = Get-VDSwitch -Name VDSwitch
# Setup some new objects to apply our settings to (reused for each)
$InfraTrafficResConfig = New-Object VMware.Vim.DvsHostInfrastructureTrafficResource
```

```
$InfraTrafficAllocationShares = New-Object VMware.Vim.SharesInfo
$InfraTrafficAllocationShares.Level = 'custom'
$InfraTrafficAllocation = New-Object
VMware.Vim.DvsHostInfrastructureTrafficResourceAllocation
$InfraTrafficAllocation.Limit = '-1'
$ResConfig = New-Object VMware.Vim.DVSConfigSpec

# enumerate the traffic types that are important to us
Foreach ($TrafficType in $VDS.ExtensionData.Config.InfrastructureTrafficResourceConfig)
{
 Switch ($TrafficType.Key) {
  # Check the vSAN Configuration for a share value of 100 & adjust if it is not 100
  "vsan" {
  If ($TrafficType.AllocationInfo.Shares.Shares -ne '100') {
   $Shares = '100'
   $InfraTrafficResConfig.Key = $TrafficType.Key
   $InfraTrafficAllocationShares.Shares = $Shares
   $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
   $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
   $InfraTrafficResConfig.AllocationInfo = $InfraTrafficAllocation
   $ResConfig.InfrastructureTrafficResourceConfig = $InfraTrafficResConfig
   $ResConfig.ConfigVersion=(Get-VDSwitch -Name $VDS).ExtensionData.Config.ConfigVersion
   # Apply the changes
   $VDS.ExtensionData.ReconfigureDvs($ResConfig)
  }
 }
 "vmotion" {
 # Check the vMotion Configuration for a share value of 50 & adjust if it is not 50
 If ($TrafficType.AllocationInfo.Shares.Shares -ne '50') {
  $Shares = '50'
  $InfraTrafficResConfig.Key = $TrafficType.Key
  $InfraTrafficAllocationShares.Shares = $Shares
  $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
  $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
  $InfraTrafficResConfig.AllocationInfo = $InfraTrafficAllocation
  $ResConfig.InfrastructureTrafficResourceConfig = $InfraTrafficResConfig
  $ResConfig.ConfigVersion=(Get-VDSwitch -Name $VDS).ExtensionData.Config.ConfigVersion
  # Apply the changes
  $VDS.ExtensionData.ReconfigureDvs($ResConfig)
 }
}
 # Check the VM Network Configuration for a share value of 30 & adjust if it is not 30
 "virtualmachine" {
 If ($TrafficType.AllocationInfo.Shares.Shares -ne '30') {
  $Shares = '30'
  $InfraTrafficResConfig.Key = $TrafficType.Key
  $InfraTrafficAllocationShares.Shares = $Shares
  $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
  $InfraTrafficResConfig.AllocationInfo = $InfraTrafficAllocation
  $ResConfig.InfrastructureTrafficResourceConfig = $InfraTrafficResConfig
  $ResConfig.ConfigVersion=(Get-VDSwitch -Name $VDS).ExtensionData.Config.ConfigVersion
  # Apply the changes
  $VDS.ExtensionData.ReconfigureDvs($ResConfig)
 }
}
"management" {
 # Check the Management Configuration for a share value of 20 & adjust if it is not 20
 If ($TrafficType.AllocationInfo.Shares.Shares -ne '19') {
  $Shares = '19'
  $InfraTrafficResConfig.Key = $TrafficType.Key
  $InfraTrafficAllocationShares.Shares = $Shares
  $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
  $InfraTrafficAllocation.Shares = $InfraTrafficAllocationShares
  $InfraTrafficResConfig.AllocationInfo = $InfraTrafficAllocation
  $ResConfig.InfrastructureTrafficResourceConfig = $InfraTrafficResConfig
  $ResConfig.ConfigVersion=(Get-VDSwitch -Name $VDS).ExtensionData.Config.ConfigVersion
  # Apply the changes
  $VDS.ExtensionData.ReconfigureDvs($ResConfig)
 }
}}
```

Setting Static Routes for Layer 3 vSAN Routing

In vSAN configurations that use Layer 3 networking, static routing is typically required. This is because vSAN uses the Default TCP/IP stack, the same as the Management VMkernel interface, and alternate gateways are not supported for interfaces that use the Default TCP/IP stack. This is not unique to vSAN and is address in more depth in KB article 2010877.

Because vSAN does not have its own TCP/IP stack, when Layer 3 addressing is used, static routes must be created on each ESXi host.

Stretched Cluster configurations are the most obvious deployments that have a requirement for Layer 3 routing. This is because hosts in the Preferred and Secondary sites must communicate with the vSAN Witness Host over Layer 3. vSAN also supports traditional configurations with Layer 3 networking, but they are less common.

Setting a static route on a vSphere host is typically set from the command line but can also be accomplished from PowerCLI.

```
PS /> New-VMHostRoute "sc1.scdemo.local" -Destination 192.168.109.0 -Gateway
192.168.110.0 -PrefixLength 24 -Confirm:$False

Destination          Gateway
-----------          -------
192.168.109.0/24     192.168.110.0
```

The above command will set a static route on a single node. Stretched Clusters typically have several nodes per site. These sites typically have different routing requirements because Stretched Clusters are required to communicate with the vSAN Witness Host over different uplinks to the location where the vSAN Witness Host is.

It would be more efficient to configure the static routes for all hosts in each site from PowerCLI. The easiest way to do this, is query the cluster for the different Preferred and Non-Preferred Fault Domains, and apply routing accordingly:

```
PS /> $PreferredFD = (Get-VsanClusterConfiguration -Cluster
"vSAN").PreferredFaultDomain
PS /> $NonPreferredFD = Get-Cluster | Get-VsanFaultDomain | Where-Object {$_.Name -ne
$PreferredFD}
PS /> $PreferredFD | Get-VMHost | New-VMHostRoute -Destination 192.168.109.0 -Gateway
192.168.110.0 -PrefixLength 24 -Confirm:$false
Destination          Gateway
-----------          -------
192.168.109.0/24     192.168.110.0
192.168.109.0/24     192.168.110.0
192.168.109.0/24     192.168.110.0

PS /> $NonPreferredFD | Get-VMHost | New-VMHostRoute -Destination 192.168.109.0 -
Gateway 192.168.110.253 -PrefixLength 24 -Confirm:$false

Destination          Gateway
-----------          -------
192.168.109.0/24     192.168.110.253
192.168.109.0/24     192.168.110.253
192.168.109.0/24     192.168.110.253
```

Setting a $PreferredFD variable to the Preferred Fault Domain and the $NonPreferredFD variable to the alternate, allows a single New-VMHostRoute command to each groups of hosts.

Tagging a vSAN Interface for vSAN Witness Traffic

For 2 Node configurations, vSAN 6.5 introduced the ability to use an alternate interface for traffic destined to communicate with a vSAN Witness Host.

The Witness Traffic Separation (WTS) feature enables the ability to directly connect 2 Nodes for vSAN data communication while communicating with the vSAN Witness Host on an alternate interface.

Upon release, support was backported to vSAN 6.2 (vSphere 6.0 U3 or higher) as well. Full Stretched Cluster support for WTS.

Traffic is tagged as “witness” for the alternate interface to communicate with the vSAN Witness Host. The Configuration Assist Wizard introduced the ability to manage this through the UI, but with the limitation that the VMkernel interface could only be backed by a vSphere Distributed Switch. Customers have largely configured “witness” traffic from the ESXi host command line.

PowerCLI can accomplish this easily with a simple one-liner:

```
PS /> Get-VMHost -Name “sc1.scdemo.local” | Get-EsxCli  -v2 | % {
$_.vsan.network.ip.add.Invoke(@{traffictype='witness';interfacename='vmk0'})}
PS />
```

In the example, vmk0 has “witness” traffic added to vmk0. Tagging “witness” traffic for all nodes in a cluster is just as easy:

```
PS /> Get-Cluster -Name “vSAN” | Get-VMHost | Get-EsxCli  -v2 | % {
$_.vsan.network.ip.add.Invoke(@{traffictype='witness';interfacename='vmk0'})}
PS />
```

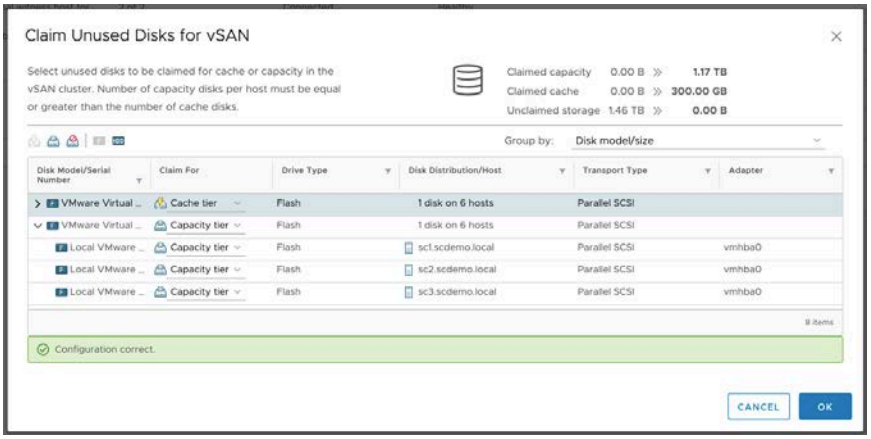
Claiming Disks on vSAN Hosts

Early releases of vSAN provided the ability to automatically claim disks for use by a vSAN host based on their eligibility. Disks could not have existing partitions and had to be presented locally to the host.

There had to be at least one flash device and one or more spinning drives in vSAN 5.5. It was easy to determine which drives were used for which requirement of cache or capacity. This was because vSAN only supported a Hybrid configuration in vSAN 5.5.

With the introduction of All-Flash support in vSAN 6.0, the auto-claim process became a bit more difficult due to all the vSAN devices being flash devices. In vSAN 6.6, the auto-claim setting was deprecated.

Despite devices no longer being automatically claimed, the vSAN Cluster Wizard will still attempt to categorize disks for either the cache or capacity use.



PowerCLI can also be used to determine which disks are eligible for vSAN, which disks are flash devices, as well as the size of the devices.

Listing the devices that are eligible for use with vSAN:

```
Get-VMHost -Name $VMHost | Get-VMHostHba | Get-ScsiLun | Where-Object
{$_.VsanStatus -eq “Eligible”}
```

The output seen will look something like this for a single host in the same cluster as above:

```
CanonicalN ConsoleDeviceName                LunType         CapacityGB
ame
---------- -----------------                -------         ----------
naa.600... /vmfs/devices/disks/naa.600... disk                 50.000
naa.600... /vmfs/devices/disks/naa.600... disk                200.000
```

Using “| format-list” (read ‘pipe format-list’), additional properties of each device can be seen as well. Format-List can be abbreviated as “fl”. Below is an abbreviated representation of the additional properties:

```
PS /> Get-VMHost -Name "sc1.scdemo.local" | Get-VMHostHba | Get-ScsiLun | Where-
Object {$_.VsanStatus -eq "Eligible"} |format-list

Id                  : HostSystem-host-13/naa.6000c291952813a5f8ff33afa05d16d0
CanonicalName       : naa.6000c291952813a5f8ff33afa05d16d0
RuntimeName         : vmhba0:C0:T1:L0
ConsoleDeviceName   : /vmfs/devices/disks/naa.6000c291952813a5f8ff33afa05d16d0
CapacityMB          : 51200
CapacityGB          : 50
HostId              : HostSystem-host-13
VMHostId            : HostSystem-host-13
VMHost              : sc1.scdemo.local
IsLocal             : True
IsSsd               : True
VsanStatus          : Eligible
ExtensionData       : VMware.Vim.HostScsiDisk
```

```
Id                  : HostSystem-host-13/naa.6000c2917d44723a60768b58ba3b43f6
CanonicalName       : naa.6000c2917d44723a60768b58ba3b43f6
RuntimeName         : vmhba0:C0:T2:L0
ConsoleDeviceName   : /vmfs/devices/disks/naa.6000c2917d44723a60768b58ba3b43f6
CapacityMB          : 204800
CapacityGB          : 200
HostId              : HostSystem-host-13
VMHostId            : HostSystem-host-13
VMHost              : sc1.scdemo.local
IsLocal             : True
IsSsd               : True
VsanStatus          : Eligible
ExtensionData       : VMware.Vim.HostScsiDisk
```

Properties such as “IsSsd” and “CapacityGB” can be used to determine whether a device is a flash device or magnetic device, and if it is above a predetermined capacity. Typically cache devices are smaller than capacity devices.

An array variable can be created and looped through to determine the number of flash devices or magnetic devices, as well as add each to separate arrays for cache devices and capacity devices. Using these different arrays will make it easy to add disk groups to a vSAN node.

```
$VsanHostDisks = Get-VMHost -Name $VMHost | Get-VMHostHba | Get-
ScsiLun | Where-Object {$_.VsanStatus -eq “Eligible”}
```

Also, remember that a vSAN Disk Group requires at least 2 devices, 1 cache and 1 capacity, so there is no need to proceed if there are less than 2 devices.

Here is an example of that:

- Places SSD’s (up to 800GB) in a Cache array
- Places SSD’s larger than 800GB or Non-SSD’s into a Capacity array

```
If ($VsanHostDisks.Count -gt 1) {
   $CacheDisks = @()
   $CapacityDisks = @()
   Foreach ($VsanDisk in $VsanHostDisks) {
      If ($VsanDisk.IsSsd -eq $true -and $VsanDisk.CapacityGB -lt
“801”) {
           $CacheDisks +=$VsanDisk
      } else {
           $CapacityDisks +=$VsanDisk
    }}}
```

It is also important to understand:

- vSAN disk groups contain 1 cache device
- vSAN disk groups may not contain more than 7 capacity devices
- VMware recommends a balanced disk group configuration

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

The number of cache devices returned will determine how many disk groups are created. It may or may not be possible to create balanced disk groups, depending on whether the overall number of capacity devices.

The switch function allows different instructions to be executed based on the value of a variable. Based on the count of cache disks, the disk group size could be balanced.

```
Switch ($CacheDisks.Count) {
  “1” { $MaxGroup = 7 }

  “2” {
   $MaxGroup = [math]::floor($CapacityDisks.Count/$CacheDisks.Count)}

  “3” {
   $MaxGroup = [math]::floor($CapacityDisks.Count/$CacheDisks.Count)}

  “4” {
   $MaxGroup = [math]::floor($CapacityDisks.Count/$CacheDisks.Count)}

  “5” {
   $MaxGroup = [math]::floor($CapacityDisks.Count/$CacheDisks.Count)}
```

If the number of capacity drives divided by the cache drive count does not result in a whole number some drives would not be consumed using this code.

The above code isn’t particularly efficient, and could be condensed:

```
Switch ($CacheDisks.Count) {
  “1” { $MaxGroup = 7
  }

  {($_ -gt 1) -and ($_ -lt 6)}
  {
   $MaxGroup = [math]::floor($CapacityDisks.Count/$CacheDisks.Count)
  }
```

Also, if the result is not a whole number, some drives would not be consumed.

Putting capacity disks into a grouped object will help when creating the disk groups.

```
$counter = [pscustomobject] @{Value =0}

$DiskGroups = $CapacityDisks | Group-Object -Property {
[math]::Floor($counter.Value++/$MaxGroup)}
```

The $DiskGroups array takes the $CapacityDisks array and groups each $DiskGroup(x) entry by the $MaxGroup size.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Setting the counter up as a custom PowerShell object allows the $DiskGroup array items to be incremented automatically as the disks are returned to the array.

The resulting output for a host with 5 cache devices and 10 capacity devices, looks something like this:

```
PS /> $DiskGroups

Count Name         Group
----- ----                    -----
    2 0            {naa.25aaf2020bdf3adfd6da06, naa.883ed1c01e644b8cedd5ab}
    2 1            {naa.2b62e4410bf13ac63865f1, naa.b86b90c1ff3afaa39db642}
    2 2            {naa.573477873e4e9d70ab9467, naa.fae008df33e152e43d0c15}
    2 3            {naa.b9f58475c012b6d5c9f40a, naa.b67cc050ce199c52e31bf5}
    2 4            {naa.64e00356e3fe0a5d508830, naa.22bcd9e7cdd02c321a9be1}
```

Creating a balanced disk group configuration for this host is as easy as enumerating each of the cache disks with one of the $DiskGroups entries.

```
$i=0
Foreach ($CacheDisk in $CacheDisks) {

   New-VsanDiskGroup -VMHost $VMHost -SsdCanonicalName $CacheDisk -
DataDiskCanonicalName $DiskGroups[$i].Group
    $i = $i+1
    Write-Host "Adding Disk Group"$i
}
```

The variable $i is set to 0 to and incremented after the disk group is created. This ensures that the $DiskGroup[$i] entry increments and the next set of capacity devices are added to the next disk group.

This resulting code looks something like this:

```
$VsanHostDisks = Get-VMHost -Name HOSTNAME | Get-VMHostHba | Get-ScsiLun |
Where-Object {$_.VsanStatus -eq "Eligible"}

If ($VsanHostDisks.Count -gt 1) {
  $CacheDisks = @()
  $CapacityDisks = @()

  Foreach ($VsanDisk in $VsanHostDisks) {
    If ($VsanDisk.IsSsd -eq $true -and $VsanDisk.CapacityGB -lt "51") {
        $CacheDisks +=$VsanDisk
    } else {
        $CapacityDisks +=$VsanDisk
    }
  }
}

$counter = [pscustomobject] @{ Value = 0 }

Switch ($CacheDisks.Count) {
  "1" {
        Write-Host "Creating 1 Disk Group because there is only 1 cache
device "
```

```
        $MaxGroup = 7
      }
 {($_ -gt 1) -and ($_ -lt 6)} {
        Write-Host "Creating 2 Disk Groups"
        $MaxGroup = [math]::floor($CapacityDisks.Count / $CacheDisks.Count)
      }
}

$DiskGroups = $CapacityDisks | Group-Object -Property {
[math]::Floor($counter.Value++ / $groupSize) }

$i=0
Foreach ($CacheDisk in $CacheDisks) {
    New-VsanDiskGroup  -VMHost $VMHost -SsdCanonicalName $CacheDisk -
DataDiskCanonicalName $DiskGroups[$i].Group
    $i = $i+1
    Write-Host "Adding Disk Group"$i
}
```

This would have to be run for every host in the vSAN cluster. It would be easier to put this into a PowerShell function that could be called for each node in the cluster.

An example function for creating disk groups:

```
Function Add-VsanHostDiskGroup {

  # Set our Parameters
  [CmdletBinding()]Param(
  [Parameter(Mandatory=$True)][string]$VMHost,
  [Parameter(Mandatory = $true)][Int]$CacheMax
  )

  # Get all of the local disks that are eligible for vSAN use
  $VsanHostDisks = Get-VMHost -Name $VMHost | Get-VMHostHba | Get-ScsiLun |
Where-Object {$_.VsanStatus -eq "Eligible"}

  # There must be at least 2 disks
  # Need to add a check to make sure at least 1 flash and 1 capacity
  If ($VsanHostDisks.Count -gt 1) {
    $CacheDisks = @()
    $CapacityDisks = @()

    # Enumerate through each of the disks.
    Foreach ($VsanDisk in $VsanHostDisks) {
      # Device is tagged as SSD and less than the max size? It is a cache
device
      If ($VsanDisk.IsSsd -eq $true -and $VsanDisk.CapacityGB -lt
$CacheMax) {
        $CacheDisks +=$VsanDisk
      } else {
        $CapacityDisks +=$VsanDisk
      }
    }
  }

  $counter = [pscustomobject] @{ Value = 0 }

Switch ($CacheDisks.Count) {
  "1" {
```


VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

```
    Write-Host "Creating 1 Disk Group because there is only 1 cache device"
    $MaxGroup = 7
  }

    {($_ -gt 1) -and ($_ -lt 6)} {
    Write-Host "Creating 2 Disk Groups"
    $groupSize = [math]::floor($CapacityDisks.Count / $CacheDisks.Count)
    }
  }
 $DiskGroups = $CapacityDisks | Group-Object -Property {
[math]::Floor($counter.Value++ / $groupSize) }

 $i=0
 Foreach ($CacheDisk in $CacheDisks) {

   # Create a new Disk Group
   New-VsanDiskGroup  -VMHost $VMHost -SsdCanonicalName $CacheDisk -
DataDiskCanonicalName $DiskGroups[$i].Group

   $i = $i+1
   Write-Host "Adding Disk Group"$i
 }
}
```

This function could then be easily executed for each host in a cluster.

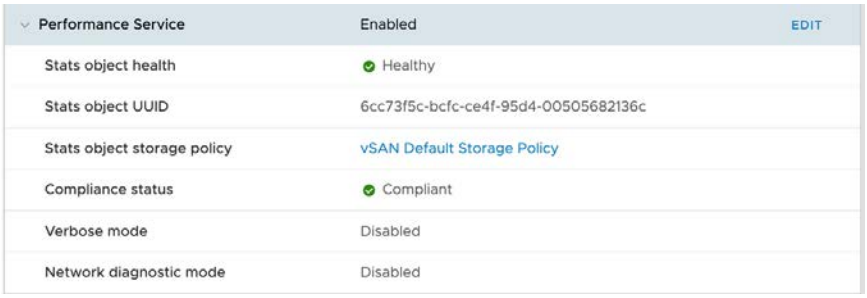
```
$Cluster = Get-Cluster -Name “ClusterName”

If ($Cluster.VsanEnabled) {

  Foreach ($ESXHost in ($Cluster | Get-VMHost)){
    Add-VsanHostDiskGroup -VMHost $ESXHost -CacheMax 400
  }
}
```

- Check to see if vSAN is enabled on the cluster.
- Use a Foreach to loop through all of the hosts in the cluster
- Execute the Add-VsanHostDiskGroup function for any eligible devices on the current host.

Remember that there is no hardcoded value for the number of hosts in the cluster, and no hardcoded value for the number of disks on each host. This example will work for vSAN clusters of any size.

vSAN Performance Service

When creating a new vSAN cluster in vSAN 6.7 or higher using the vSAN Cluster Wizard or vSphere Cluster Quickstart, the Performance Service is automatically enabled.



When using older versions of vSAN or are creating a cluster using the vSAN Management API, the Performance Service is not enabled automatically.

Enabling the Performance Service is relatively easy with PowerCLI using the Get/Set-VsanClusterConfiguration cmdlets.

```
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Set-
VsanClusterConfiguration -PerformanceServiceEnabled $true

Cluster             VsanEnabled  IsStretchedCluster   Last HCL Updated
-------             -----------  ------------------   ----------------
vSAN                True         True                 1/16/19 11:33:00 AM
```

This will enable the vSAN Performance Service with vSAN's default storage policy. If an alternate policy is desired, the -StoragePolicy parameter may be used.

```
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Set-
VsanClusterConfiguration -PerformanceServiceEnabled $true -StoragePolicy
(Get-SpbmStoragePolicy -Name “Alternate Policy”)

Cluster             VsanEnabled  IsStretchedCluster   Last HCL Updated
-------             -----------  ------------------   ----------------
vSAN                True         True                 1/16/19 11:33:00 AM
```

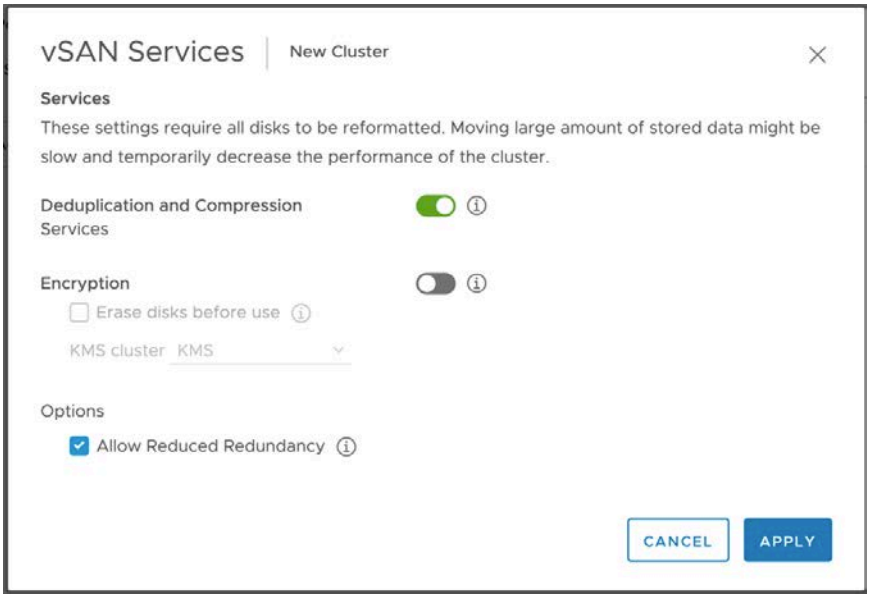
The storage policy could be put in a variable beforehand as well

```
Ps /> $Policy = Get-SpbmStoragePolicy -Name “Alternate Policy”
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Set-
VsanClusterConfiguration -PerformanceServiceEnabled $true -StoragePolicy
$Policy

Cluster             VsanEnabled  IsStretchedCluster   Last HCL Updated
-------             -----------  ------------------   ----------------
vSAN                True         True                 1/16/19 11:33:00 AM
```

Deduplication and Compression

Deduplication and Compression was introduced with the release of vSAN 6.2. This service gives administrators the ability to enable the deduplication and compression of data on All-Flash vSAN clusters with Advanced or Enterprise licensing.

Enabling Deduplication and Compression
Deduplication and Compression is typically enabled in the vSphere Client or vSphere Web Client in the vSAN Services UI. Deduplication and Compression can be enabled upon cluster creation, or it may be enabled on an existing cluster.



Deduplication and Compression settings for a vSAN cluster can be configured using the Get/Set-VsanClusterConfiguration cmdlets.

```
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Select-Object Name,
VsanEnabled, SpaceEfficiencyEnabled

Name VsanEnabled SpaceEfficiencyEnabled
---- ----------- ----------------------
vSAN        True                  False
```

To set SpaceEfficiencyEnabled the vSAN Cluster's configuration must be passed to the Set-VsanClusterConfiguration cmdlet.

```
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Set-
VsanClusterConfiguration -SpaceEfficiencyEnabled $true

Cluster              VsanEnabled  IsStretchedCluster   Last HCL Updated
-------              -----------  ------------------   ----------------
vSAN                 True         True                 1/16/19 11:33:00 AM
```

After the process of enabling SpaceEfficiency, Get-VsanClusterConfiguration will report that it has been enabled.

```
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Select-Object Name,
VsanEnabled, SpaceEfficiencyEnabled
```

```
Name VsanEnabled SpaceEfficiencyEnabled
---- ----------- ----------------------
vSAN        True                    True
```

In cases where the cluster is a 2 or 3 Node vSAN cluster, remember to include the -AllowReducedRedundancy parameter.

```
Ps /> Get-VsanClusterConfiguration -Cluster "vSAN" | Set-
VsanClusterConfiguration -SpaceEfficiencyEnabled $true -
AllowReducedRedundancy $true

Cluster              VsanEnabled  IsStretchedCluster   Last HCL Updated
-------              -----------  ------------------   ----------------
vSAN                 True         True                 1/16/19 11:33:00 AM
```

Deduplication and Compression can also be set using Get-VsanView and the VsanVcClusterConfigSystem Managed Object. While this isn't the easiest method, it is still good to be familiar with when combined with vSAN Encryption.

This is because the process of enabling vSAN Encryption simultaneously with Deduplication and Compression requires only a single on-disk format change.

```
# Get the Datacenter Object
$Datacenter = Get-Datacenter -Name "Datacenter"

# Create a new Cluster
$Cluster = New-Cluster -Name “vSAN” -VsanEnabled -Location $Datacenter

# Setup the VsanVcClusterConfigSystem variable
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-
system"

# The DataEfficiency setting must be put in a Specification
$VsanDataEfficiencyConfig = New-Object -TypeName
VMware.Vsan.Views.VsanDataEfficiencyConfig
# Compression must be set
$VsanDataEfficiencyConfig.CompressionEnabled = $true
# Deduplication must be set
$VsanDataEfficiencyConfig.DedupEnabled = $true

# The vSAN Config must be set in a specification
$VsanConfig = New-Object -Type VMware.Vsan.Views.VimVsanReconfigSpec
# Set Reduced Redundancy for 2 or 3 node configurations
$VsanConfig.AllowReducedRedundancy = $true
# Set the Data Encryption Configuration
$VsanConfig.DataEfficiencyConfig = $VsanDataDfficiencyConfig
# Execute a Cluster Reconfiguration
$VsanVcClusterConfig.VsanClusterReconfig($Cluster.ExtensionData.MoRef,$VsanConfig)
```

The process is the same for an existing cluster that will have encryption enabled.

```
# Get the Cluster
$Cluster = Get-Cluster -Name “vSAN”

# Setup the VsanVcClusterConfigSystem variable
```

```
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-
system"

# The DataEfficiency setting must be put in a Specification
$VsanDataEfficiencyConfig = New-Object -TypeName
VMware.Vsan.Views.VsanDataEfficiencyConfig
# Compression must be set
$VsanDataEfficiencyConfig.CompressionEnabled = $true
# Deduplication must be set
$VsanDataEfficiencyConfig.DedupEnabled = $true

# The vSAN Config must be set in a specification
$VsanConfig = New-Object -Type VMware.Vsan.Views.VimVsanReconfigSpec
# Set Reduced Redundancy for 2 or 3 node configurations
$VsanConfig.AllowReducedRedundancy = $true
# Set the Data Encryption Configuration
$VsanConfig.DataEfficiencyConfig = $VsanDataDfficiencyConfig
# Execute a Cluster Reconfiguration
$VsanVcClusterConfig.VsanClusterReconfig($Cluster.ExtensionData.MoRef,$VsanConfig)
```

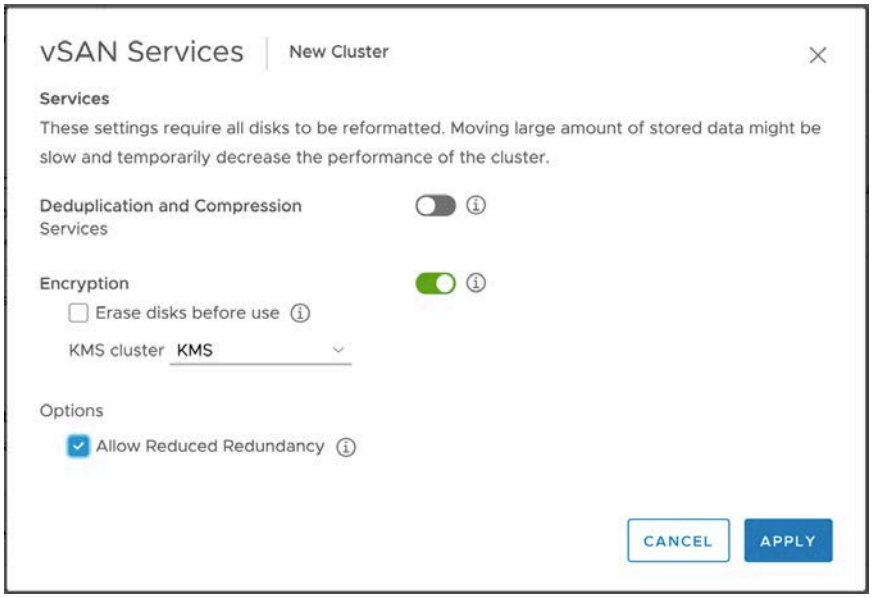
Closely looking at the code, it can be seen that Compression and Deduplication are separate data efficiency settings. At this time, enabling one, but not the other is not supported by VMware.

vSAN Encryption

Data at Rest Encryption was introduced for vSAN with the release of version 6.6. This service gives administrators the ability to encrypt all data on a vSAN cluster on both Hybrid and All-Flash Architectures.

Enabling vSAN Encryption

vSAN Encryption is typically enabled in the vSphere Client or vSphere Web Client in the vSAN Services UI. Encryption can be enabled upon cluster creation, or it may be enabled on an existing cluster.



Enabling or disabling Encryption on a vSAN Cluster can be performed using the Start-VsanEncryptionConfiguration cmdlet. This cmdlet is used to enable or disable vSAN Encryption.

Parameters that are used when enabling or disabling encryption with the Start-VsanEncryptionConfiguration cmdlet include the Cluster object, the KMS Server object, the encryption state (enabled/disabled), whether to erase disks before use, and whether or not to allow for reduced redundancy.

Here is an example set of commands used to enable vSAN Encryption:

```
PS /> $KmsCluster = Get-KmsCluster -Name "KMS"
PS /> $KmsCluster = Get-KmsCluster -Name "KMS"
PS /> Start-VsanEncryptionConfiguration -Cluster $Cluster -EncryptionEnabled $true -
EraseDisksBeforeUse $false -AllowReducedRedundancy $true -KmsCluster $KmsCluster -
Confirm:$false

Name                           State      % Complete Start Time   Finish Time
----                           -----      ---------- ----------   -----------
Reconfigure vSAN cluster       Running             0 09:04:21 AM
```

Disabling vSAN Encryption would be similarly disabled:

```
PS /> Start-VsanEncryptionConfiguration -Cluster $Cluster -EncryptionEnabled $false -
AllowReducedRedundancy $true -Confirm:$false

Name                           State      % Complete Start Time   Finish Time
----                           -----      ---------- ----------   -----------
Reconfigure vSAN cluster       Running             0 09:23:54 AM
```

Going a bit further, it would be appropriate to set AllowReducedRedundancy to $true when only 2 or 3 hosts are in the cluster. This is the default recommendation, but could be used when additional hosts are present.

```
$Cluster = Get-Cluster -Name “vSAN”
$VMHosts = $Cluster | Get-VMHost
$KmsCluster = Get-KmsCluster -Name “KMS”

# If there are less than 3 hosts then set Allow Reduced Redundancy to $true
If ($VMHosts.Count -lt 4) {
  $AllowReducedRedundancy = $true
} else {
  $AllowReducedRedundancy = $false
}

Start-VsanEncryptionConfiguration -Cluster $Cluster -EncryptionEnabled $true -
EraseDisksBeforeUse $false -AllowReducedRedundancy $AllowReducedRedundancy -KmsCluster
$KmsCluster -Confirm:$false
```

The Get-VsanView cmdlet can also be used to configure vSAN Encryption. The VsanVcClusterConfigSystem Managed Object will be used to modify a Cluster after it is created.

```
# Get the Datacenter Object
$Datacenter = Get-Datacenter -Name "Datacenter"
```

```
# Create a new Cluster
$Cluster = New-Cluster -Name “vSANCluster” -VsanEnabled -Location $Datacenter

# Setup the VsanVcClusterConfigSystem variable
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-
system"

# The DataEncryption setting must be put in a Specification
$VsanEncryptionConfig = New-Object -Type VMware.Vsan.Views.VsanDataEncryptionConfig
# Enable Encryption
$VsanEncryptionConfig.EncryptionEnabled = $true
# Grab the Provider ID for the KMS from vCenter
$VsanEncryptionConfig.KmsProviderId = (Get-KmsCluster -Name
"KMS").ExtensionData.ClusterId
# Set Erase Disks Before Use to remove residual data
$VsanEncryptionConfig.EraseDisksBeforeUse = $true

# The vSAN Config must be set in a specification
$VsanConfig = New-Object -Type VMware.Vsan.Views.VimVsanReconfigSpec
# Set Reduced Redundancy for 2 or 3 node configurations
$VsanConfig.AllowReducedRedundancy = $true
# Set the Data Encryption Configuration
$VsanConfig.DataEncryptionConfig = $VsanEncryptionConfig
# Execute a Cluster Reconfiguration
$VsanVcClusterConfig.VsanClusterReconfig($Cluster.ExtensionData.MoRef,$VsanConfig)
```

The process is the same for an existing cluster that will have encryption enabled.

```
# Get the Cluster
$Cluster = Get-Cluster -Name “vSANCluster”

# Setup the VsanVcClusterConfigSystem variable
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-
system"

# The DataEncryption setting must be put in a Specification
$VsanEncryptionConfig = New-Object -Type VMware.Vsan.Views.VsanDataEncryptionConfig
# Enable Encryption
$VsanEncryptionConfig.EncryptionEnabled = $true
# Grab the Provider ID for the KMS from vCenter
$VsanEncryptionConfig.KmsProviderId = (Get-KmsCluster -Name
"KMS").ExtensionData.ClusterId
# Set Erase Disks Before Use to remove residual data
$VsanEncryptionConfig.EraseDisksBeforeUse = $true

# The vSAN Config must be set in a specification
$VsanConfig = New-Object -Type VMware.Vsan.Views.VimVsanReconfigSpec
# Set Reduced Redundancy for 2 or 3 node configurations
$VsanConfig.AllowReducedRedundancy = $true
# Set the Data Encryption Configuration
$VsanConfig.DataEncryptionConfig = $VsanEncryptionConfig
# Execute a Cluster Reconfiguration
$VsanVcClusterConfig.VsanClusterReconfig($Cluster.ExtensionData.MoRef,$VsanConfig)
```

As mentioned at the end of the Deduplication and Compression section, enabling Deduplication and Compression along with vSAN Encryption simultaneously is often desirable, because they both require an on-disk format change. These can easily be added together with the samples from the Deduplication and Compression section and this section:

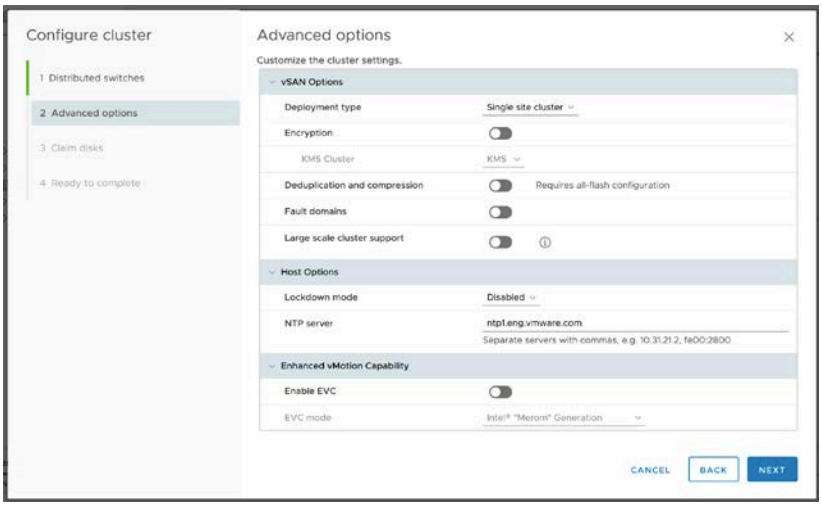
```
# Get the Cluster
$Cluster = Get-Cluster -Name “vSANCluster”
```

```
# Setup the VsanVcClusterConfigSystem variable
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-
system"

# The DataEfficiency setting must be put in a Specification
$VsanDataEfficiencyConfig = New-Object -TypeName
VMware.Vsan.Views.VsanDataEfficiencyConfig
# Compression must be set
$VsanDataEfficiencyConfig.CompressionEnabled = $true
# Deduplication must be set
$VsanDataEfficiencyConfig.DedupEnabled = $true

# The DataEncryption setting must be put in a Specification
$VsanEncryptionConfig = New-Object -Type VMware.Vsan.Views.VsanDataEncryptionConfig
# Enable Encryption
$VsanEncryptionConfig.EncryptionEnabled = $true
# Grab the Provider ID for the KMS from vCenter
$VsanEncryptionConfig.KmsProviderId = (Get-KmsCluster -Name
"KMS").ExtensionData.ClusterId
# Set Erase Disks Before Use to remove residual data
$VsanEncryptionConfig.EraseDisksBeforeUse = $true

# The vSAN Config must be set in a specification
$VsanConfig = New-Object -Type VMware.Vsan.Views.VimVsanReconfigSpec
# Set Reduced Redundancy for 2 or 3 node configurations
$VsanConfig.AllowReducedRedundancy = $true
# Set the Data Efficiency Configuration
$VsanConfig.DataEfficiencyConfig = $VsanDataEfficiencyConfig
# Set the Data Encryption Configuration
$VsanConfig.DataEncryptionConfig = $VsanEncryptionConfig
# Execute a Cluster Reconfiguration
$VsanVcClusterConfig.VsanClusterReconfig($Cluster.ExtensionData.MoRef,$VsanConfig)
```

Configuring NTP

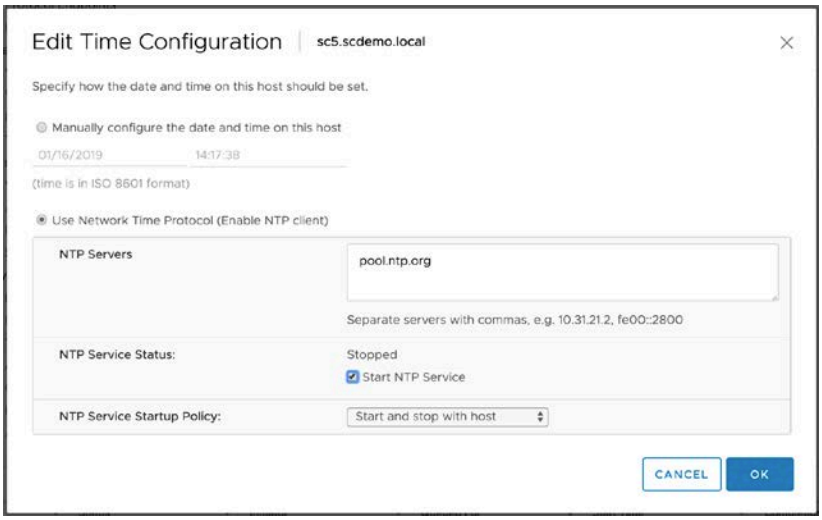
While not completely necessary for vSAN operations, it is still a best practice for vSphere hosts have to have a consistent time configuration across all the hosts in the vSAN cluster.

In vCenter 6.7 Update 1, the Cluster Quickstart will set NTP for all hosts in the cluster.



When not using the Cluster Quickstart found in vCenter 6.7 Update 1,

NTP can be set for each host individually in the vSphere Client.



Notice that the NTP service must be enabled, contains a list of one or more NTP servers, the current state of the service, the ability to start it, and the normal startup policy.

A few cmdlets will be used to perform each of these tasks:

- Add-VMHostNtpServer – NTP Server addresses
- Get/Set-VMHostFirewallException – Allowing the ESXi host to communicate with the NTP servers
- Get/Set/Start-VmHostService – Starts NTP and sets the behavior of the service.

Setting NTP on a single vSphere host:

```
PS /> $VMHost = Get-VMHost -Name "sc1.scdemo.local"
PS /> Add-VmHostNtpServer -VmHost $VMHost -NtpServer "ntp1.eng.vmware.com"
ntp1.eng.vmware.com
PS /> Get-VMHostFirewallException -VMHost $VMHost | Where-Object {$_.Name -eq "NTP
client"} | Set-VMHostFirewallException -Enabled $true

Name       Enabled IncomingPorts  OutgoingPorts  Protocols  ServiceRunning
----       ------- -------------  -------------  ---------  --------------
NTP Client True                   123            UDP        False

PS /> Get-VMHostService -VMHost $VMHost | Where-Object {$_.Key -eq "ntpd"} | Start-
VMHostService

Key         Label                   Policy     Running  Required
---         -----                   ------     -------  --------
ntpd        NTP Daemon              off        True     False

PS />
PS /> Get-VMHostService -VMHost $VMHost | Where-Object {$_.Key -eq "ntpd"} | Set-
VMHostService -Policy "Automatic"

Key             Label                         Policy     Running  Required
---             -----                         ------     -------  --------
```

```
ntpd            NTP Daemon                    automatic  True     False

PS />
```

It is a bit more efficient to set a uniform NTP configuration across an entire cluster:

```
$Cluster = Get-Cluster -Name "vSAN"
Foreach ($VMHost in ($Cluster | Get-VMHost)) {
    Add-VmHostNtpServer -VmHost $VMHost -NtpServer "ntp1.eng.vmware.com"
    Get-VMHostFirewallException -VMHost $VMHost | Where-Object {$_.Name -eq "NTP
client"} | Set-VMHostFirewallException -Enabled $true
    Get-VMHostService -VMHost $VMHost | Where-Object {$_.Key -eq "ntpd"} | Start-
VMHostService
    Get-VMHostService -VMHost $VMHost | Where-Object {$_.Key -eq "ntpd"} | Set-
VMHostService -Policy "Automatic"
}
```

Or all hosts connected to vCenter:

```
Foreach ($VMHost in Get-VMHost) {
    Add-VmHostNtpServer -VmHost $VMHost -NtpServer "ntp1.eng.vmware.com"
    Get-VMHostFirewallException -VMHost $VMHost | Where-Object {$_.Name -eq "NTP
client"} | Set-VMHostFirewallException -Enabled $true
    Get-VMHostService -VMHost $VMHost | Where-Object {$_.Key -eq "ntpd"} | Start-
VMHostService
    Get-VMHostService -VMHost $VMHost | Where-Object {$_.Key -eq "ntpd"} | Set-
VMHostService -Policy "Automatic"
}
```

Uniform NTP settings are an often-overlooked vSphere host configuration setting. The vSAN Health Check now checks for this.

Configuring vSphere HA

Another important cluster service that is typically configured with a vSAN Cluster is vSphere Availability, commonly referred to as vSphere HA.

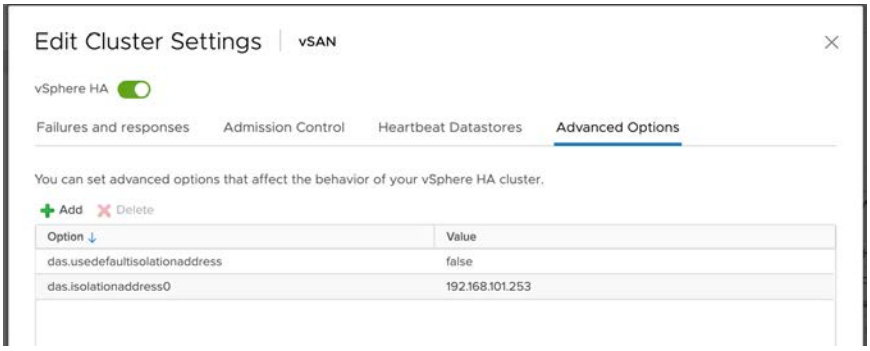
The Set-Cluster cmdlet covers much of the configuration with the use of the HAEnabled, HAAdmissionControlEnabled, and HAIsolationResponse parameters.

```
PS /> Get-Cluster -Name “vSAN” | Set-Cluster -HAEnabled:$true -
HAAdmissionControlEnabled:$true -HAIsolationResponse “Powered Off”
```

There are some advanced settings that also need to be set for vSAN, but they are not part of the Set-Cluster cmdlet.

When used in conjunction with vSAN, vSphere HA uses the vSAN data network for its heartbeats. To configure this properly the vSphere HA – Advanced Options UI is used, and two additional settings are added.

- das.usedefaultisolationaddress
- das.isolationaddress0



While using PowerCLI is great to configure vSphere HA, it would be unfortunate if going into the UI was still required to add these advanced settings. The New-AdvancedSetting cmdlet can be used to apply these to the vSAN Cluster's HA configuration.

```
PS /> $Cluster = Get-Cluster -Name "vSAN"
PS / > New-AdvancedSetting -Entity $Cluster -Type ClusterHA -Name
"das.isolationaddress0" -Value "192.168.101.253" -Confirm:$false

Name                 Value                Type                 Description
----                 -----                ----                 -----------
das.isolationaddr... 192.168.101.253      ClusterHA

PS /Users/jase> New-AdvancedSetting -Entity $Cluster -Type ClusterHA -Name
"das.usedefaultisolationaddress" -Value "False" -Confirm:$false

Name                 Value                Type                 Description
----                 -----                ----                 -----------
das.usedefaultiso... False                ClusterHA

PS />
```

Configuring vSphere DRS

While not as important, and not available in all editions of vSphere, an administrator may desire to configure vSphere Distributed Resource Scheduling (DRS) as part of a cluster configuration.

The Set-Cluster cmdlet can configure the basic DRS settings using the DrsEnabled and DrsAutomationLevel paramters.

```
PS /> Get-Cluster -Name “vSAN” | Set-Cluster -DrsEnabled:$true -DrsAutomationLevel
”FullyAutomated”
```

Advanced settings for can be configured using the New-AdvancedSetting cmdlet as well.

Configuring Guest TRIM & UNMAP Support

Data is thin provisioned when written to a vSAN data store by default. As a virtual machine's hard disks grow to their provisioned capacity, additional capacity is consumed on a vSAN datastore. When data is deleted from that virtual machine, the capacity is removed inside the guest's disks, but not removed from the vSAN datastore.

With the release of vSAN 6.7 Update 1, guest level TRIM & UNMAP support was added. The addition of Guest TRIM & UNMAP support on vSAN allows vSAN datastore capacity to be reclaimed by vSAN.

Guest TRIM & UNMAP support must be enabled for a vSAN cluster, and guests must meet some requirements to be able to work with Guest TRIM & UNMAP on vSAN.

- Windows guests – Minimum VM Hardware version 11
- Linux guests – Minimum VM Hardware version 13
- *disk.scsiUnmapAllowed* flag – not set to false
- Guest OS – must be able to identify virtual disk as thin
- Enabled – At vSAN Cluster level
- VM's – Must be power cycled after setting to use

More information can be found about Guest TRIM & UNMAP on StorageHub in Space Efficiency Technologies for vSAN.

Initially this feature could only be enabled using the Ruby vSphere Console (RVC). This required logging into the RVC, navigating to the Cluster object in the RVC, and then executing the **vsan.unmap_support** action.

PowerCLI 11.2 introduced the ability to configure this setting using the Get/Set-VsanClusterConfiguration cmdlets. It is even easier using PowerCLI to set, and report on, this setting in a vSAN 6.7 Update 1 cluster.

To determine if a vSAN cluster has Guest TRIM & UNMAP enabled, the following code may be used:

```
PS /> Get-VsanClusterconfiguration -Cluster (Get-Cluster -Name "vSAN") | Select-Object
Name, guestTrimUnmap

Name               GuestTrimUnmap
-----              --------------
vSAN                        False
```

Enabling Guest TRIM & UNMAP is just as easy:

```
PS /> Get-VsanClusterconfiguration -Cluster (Get-Cluster -Name "vSAN") |
Set-VsanClusterConfiguration -guestTrimUnmap $true

Name               GuestTrimUnmap
-----              --------------
vSAN                         True
```

Operational Recipes

A few sample 'Recipes' are included in this document to detail the process of how one would go about putting together PowerCLI scripts for vSAN together.

These will only be code snippets included in this document. Each recipe will include a link to a competed sample script in the respective summary section.

> **Important Note:** The code samples included in this document are not supported by VMware. The code included is only provided as sample code for the purpose of demonstrating different tasks using PowerCLI.

Host Maintenance & Tasks

vSAN is component of vSphere. Updating and maintaining vSAN is typically as easy as updating vSphere.

The core function of updating a vSphere host, where vSAN is enabled or not, is putting that host in Maintenance Mode. PowerCLI has had the capability of doing this for quite a while with the Set-VMHost command:

```
PS /> Set-VMHost -VMHost "sc1.scdemo.local" -State "Maintenance"

Name                 ConnectionState PowerState
----                 --------------- ----------
sc1.scdemo.local     Maintenance     PoweredOn
```

The default Maintenance Mode for vSAN Clusters is "Ensure Accessibility" and should not have to be specified. It can however be specified to guarantee the Maintenance Mode choice:

```
PS /> Set-VMHost -VMHost "sc1.scdemo.local" -State "Maintenance" -
VsanDataMigrationMode "EnsureAccessibility"

Name                 ConnectionState PowerState
----                 --------------- ----------
sc1.scdemo.local     Maintenance     PoweredOn
```

Taking a host out of Maintenance Mode is just as easy:

```
PS /> Set-VMHost -VMHost "sc1.scdemo.local" -State "Connected"

Name                 ConnectionState PowerState
----                 --------------- ----------
sc1.scdemo.local     Connected       PoweredOn
```

Patch Management with Update Manager

Patch management is handled by VMware Update Manager. PowerCLI does have Update Manager support, but it is only available in PowerShell, and not currently in PowerShell Core. These cmdlets are

currently only available on Windows systems using PowerShell and PowerCLI. Baselines are associated.

The Get-Baseline cmdlet will return the baselines available on the Update Manager instance:

```
PS C:\> Get-Baseline -TargetType Host

Name                 Description          Id  Type  Targt LastUpdateTime       Num
----                 -----------          --  ----  ----- --------------       -----
BNX                  Updated BNX Drivers  10  Patch Host  1/21/2019 11:15:0... 6
VMware ESXi 6.5.0... VMware ESXi 6.5.0... 9   Patch Host  1/18/2019 4:12:41 PM 2
VMware ESXi 6.7.0... VMware ESXi 6.7.0... 8   Patch Host  1/16/2019 12:45:4... 2
Non-Critical Host... A predefined base... 3   Patch Host  1/7/2019 1:03:55 PM  132
Critical Host Pat... A predefined base... 2   Patch Host  1/7/2019 1:03:55 PM  28
```

Baseline Groups are not returned by Get-Baseline cmdlet.

For a baseline to be checked or patched against, first a baseline must be attached to a host or cluster.

To make the attach process easier, let's set our baselines to variables:

```
PS C:\> $BaselineNonCritical = Get-Baseline -Name "Non-Critical Host Patches
(Predefined)"
PS C:\> $BaselineCritical = Get-Baseline -Name " Critical Host Patches (Predefined)"
```

With the baselines set to variables, it is easy to attach these to a vSAN cluster:

```
PS C:\> $Cluster = Get-Cluster -Name "vSAN"
PS C:\> Attach-Baseline -Baseline $BaselineCritical -Entity $Cluster
PS C:\> Attach-Baseline -Baseline $BaselineNonCritical -Entity $Cluster
PS C:\>
```

It is also easy to scan the vSAN Cluster for compliance with the attached baselines:

```
PS C:\> $Cluster | Test-Compliance -UpdateType HostPatch -RunAsync

Name                           State      % Complete Start Time   Finish Time
----                           -----      ---------- ----------   -----------
Scan entity                    Running             0 11:25:39 AM
```

For a newly configured environment that does not have any workloads, all hosts could be scanned simultaneously:

```
PS C:\> Foreach ($ESXhost in ($Cluster | Get-VMHost)) { Test-Compliance -Entity
$ESXhost -UpdateType HostPatch -RunAsync}

Name                           State      % Complete Start Time   Finish Time
----                           -----      ---------- ----------   -----------
Scan entity                    Queued              0 12:00:00 AM
Scan entity                    Queued              0 12:00:00 AM
Scan entity                    Running             0 11:26:42 AM
Scan entity                    Queued              0 12:00:00 AM
Scan entity                    Queued              0 12:00:00 AM
Scan entity                    Queued              0 12:00:00 AM
```

After hosts have been scanned, the compliance with attached baselines can be reported on using Get-Compliance:

```
PS C:\> $Cluster | Get-Compliance

Entity                         Baseline                         Status
------                         --------                         ------
sc1.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc1.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc1.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc6.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc6.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc6.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc3.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc3.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc3.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc5.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc5.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc5.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc2.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc2.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc2.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc4.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc4.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc4.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
```

If the output is a bit hard to follow, it might be easier to sort on the name of the ESXi host:

```
PS C:\> $Cluster | Get-Compliance | Sort-Object -Property Entity, Baseline

Entity                         Baseline                         Status
------                         --------                         ------
sc1.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc1.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc1.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc2.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc2.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc2.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc3.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc3.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc3.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc4.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc4.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc4.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc5.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc5.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc5.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc6.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc6.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc6.scdemo.local               Critical Host Patches (Predefined)  Compliant
```

Updates are typically only applied to hosts that are not compliant.

By piping the results through Where-Object, it is easy to only return those that are NotCompliant.

It will likely also be beneficial to sort the output on the ESXi hostname for clarity.

```
PS C:\> $Cluster | Get-Compliance | Where-Object {$_.Status -eq "NotCompliant"} | Sort-
Object -Property Entity
```

```
Entity                         Baseline                          Status
------                         --------                          ------
sc1.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc2.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc3.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc4.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc5.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
sc6.scdemo.local               Non-Critical Host Patches (Prede... NotCompliant
```

Above, each of hosts are not compliant with the Non-Critical Host Patches baseline.

The Update-Entity cmdlet is used to remediate a cluster or host. A few things to consider though, are that the operation will require a confirmation, and the cmdlet requires some time. The -RunAsync option will be necessary to prevent an error being returned:

```
PS C:\> $Cluster | Update-Entity -Baseline $BaselineNonCritical -Confirm:$false -
RunAsync

Name                           State      % Complete Start Time   Finish Time
----                           -----      ---------- ----------   -----------
Remediate entity               Queued               0 12:00:00 AM
```

Only hosts that are not compliant will be updated with the baseline. Update Manager will cycle through each of the hosts and update them.

```
PS C:\> $Cluster | Get-Compliance | Sort-Object -Property Entity

Entity                         Baseline                          Status
------                         --------                          ------
sc1.scdemo.local               Non-Critical Host Patches (Prede... Compliant
sc1.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc1.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc2.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc2.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc2.scdemo.local               Non-Critical Host Patches (Prede... Compliant
sc3.scdemo.local               Non-Critical Host Patches (Prede... Compliant
sc3.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc3.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc4.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc4.scdemo.local               Non-Critical Host Patches (Prede... Compliant
sc4.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc5.scdemo.local               Critical Host Patches (Predefined)  Compliant
sc5.scdemo.local               Non-Critical Host Patches (Prede... Compliant
sc5.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc6.scdemo.local               VMware ESXi 6.7.0 U1 (Patch ESXi... Compliant
sc6.scdemo.local               Non-Critical Host Patches (Prede... Compliant
sc6.scdemo.local               Critical Host Patches (Predefined)  Compliant
```

Installing a VIB on a vSAN Host

Different versions of vSphere often have different driver requirements for specific hardware. Looking at this from a vSAN view, an administrator often times needs to install an updated driver for a storage controller.

vSAN is still part of vSphere and relies on other pieces of the stack to work optimally. Driver updates for network cards is another item to consider as well.

Whether they are storage controller drivers, network card drivers, or other packages, are typically installed using the vSphere Installable Bundle (VIB) specification. VIBs are typically provided by vendors in a .vib file included in a zipped file. Sometimes the vib will be properly signed and sometimes not. Sometimes the vib will be included in an "offline bundle" zip file.

David Stamen, a Technical Marketing Engineer on the vSphere team covers how to install a vib on an ESXi host: https://davidstamen.com/2016/03/03/using-powercli-to-install-host-vibs/

Breaking down his code, we can see that he's using the Get-EsxCli cmdlet to connect to an instance of the **esxcli** command line utility on each connected vSphere host, setting some parameters for the vib, invoking the software install process, and then verifying the vib was installed properly.

```
# Define Variables
$Cluster = Get-Cluster -Name "vSAN"
$vibpath = "/vmfs/volumes/NFS01/VIB/cisco/scsi-fnic_1.6.0.24-1OEM.600.0.0.2494585.vib"

# Get each host in specified cluster that meets criteria
Get-VMhost -Location $Cluster | where { $_.PowerState -eq "PoweredOn" -and
$_.ConnectionState -eq "Connected" } | foreach {

    Write-host "Preparing $($_.Name) for ESXCLI" -ForegroundColor Yellow

    $ESXCLI = Get-EsxCli -VMHost $_ -V2

    # Install VIBs
    Write-host "Installing VIB on $($_.Name)" -ForegroundColor Yellow

                    # Create Installation Arguments
                    $insParm = @{
                              viburl = $vibpath
                              dryrun = $false
                              nosigcheck = $true
                              maintenancemode = $false
                              force = $false
                    }

          $action = $ESXCLI.software.vib.install.Invoke($insParm)

    # Verify VIB installed successfully
    if ($action.Message -eq "Operation finished successfully."){
      Write-host "Action Completed successfully on $($_.Name)"
    } else {
      Write-host $action.Message
  }
}
```

- Notice that the vib path is on an NFS datastore.
- Also notice that nosigcheck is $true – Some vibs are not signed by the distributing vendor. These will cause issues if using SecureBoot.

Rebooting a vSAN Host
Rebooting a vSAN host is as simple as putting a host in maintenance mode and executing a reboot operation.

Putting a vSAN Host in Maintenance Mode using Set-VMHost:

```
PS /> Set-VMHost -VMHost sc1.scdemo.local -State Maintenance -VsanDataMigrationMode
"EnsureAccessibility"

Name                 ConnectionState PowerState NumCpu Version
----                 --------------- ---------- ------ -------
sc1.scdemo.local     Maintenance     PoweredOn       2   6.7.0
```

The Restart-VMHost cmdlet is then used to reboot the host. RunAsync is used because the reboot process will take longer than the command timeout. And Confirm:$false is used to prevent a prompt.

```
PS /> Restart-VMHost sc1.scdemo.local -RunAsync -Confirm:$false

Name                           State      % Complete Start Time   Finish Time
----                           -----      ---------- ----------   -----------
RebootHost_Task
```

When the host has rebooted and become available again, we can set the state to Connected to exit Maintenance Mode with Set-VMHost:

```
PS /> Set-VMHost -VMHost sc1.scdemo.local -State Connected

Name                 ConnectionState PowerState NumCpu Version
----                 --------------- ---------- ------ -------
sc1.scdemo.local     Connected       PoweredOn       2   6.7.0
```

Putting these together would look something like this:

```
# Get the host object
$VMHost = "sc1.scdemo.local"

# Put the host in maintenance mode
Set-VMHost $VMHost -State Maintenance -VsanDataMigrationMode "EnsureAccessibility"

# Once in maintenance mode, restart the host
Restart-VMHost $VMHost -RunAsync -Confirm:$false

# Give status updates while waiting to reboot
Write-Host "Waiting on $VMHost to reboot"
While ((Get-VMHost $VMHost).ConnectionState -ne "NotResponding") {
    Start-Sleep 1
    Write-Host "." -NoNewLine
}
Write-Host ""

# Give status updates while waiting for the host to come back online
Write-Host "$VMHost is rebooting"

While ((Get-VMhost $VMHost).ConnectionState -eq "NotResponding") {
    Start-Sleep 2
    Write-Host "." -NoNewLine
}
Write-Host ""
Write-Host "$VMHost is online"
Set-VMHost $VMHost -State Connected
```

The output looks something like this:

```
Name                 ConnectionState PowerState NumCpu Version
----                 --------------- ---------- ------ -------
sc1.scdemo.local     Maintenance     PoweredOn       2   6.7.0

ServerId        : /VIServer=vsphere.local\administrator@10.198.6.18:443/
State           : Success
IsCancelable    : False
PercentComplete : 100
StartTime       : 1/23/19 12:47:33 PM
FinishTime      : 1/23/19 12:47:33 PM
ObjectId        : HostSystem-host-13
Result          :
Description     : Initiate host reboot
ExtensionData   : VMware.Vim.Task
Id              : Task-task-6308
Name            : RebootHost_Task
Uid             : /VIServer=vsphere.local\administrator@10.198.6.18:443/Task=Task-task-
6308/
CmdletTaskInfo  :

Waiting on sc1.scdemo.local to reboot
.....
sc1.scdemo.local is rebooting
.....................................................
sc1.scdemo.local is online
sc1.scdemo.local     Connected       PoweredOn       2    6.7.0
```

Powering off a vSAN Cluster

Another common request is, how would an administrator power off an entire vSAN Cluster?

Why would someone want to do that? Maybe the cluster is being moved to another location or facility.

This section will focus on the steps identified in KB article 2142676.

With the exception of vCenter, all virtual machines must be shut down.

To get the VMs that have VMware Tools installed and are not vCenter, the ToolsStatus extended configuration property can be used. We'll put these in an array:

```
PS /> $VMsWithTools = Get-VM | Where-Object {($_.ExtensionData.Guest.ToolsStatus -eq
'toolsOk') -and ($_.Name -ne "VCSA")}
```

VM's without VMware Tools installed, or VM's the VM Tools in a quasi-available state can be put in a separate array:

```
PS /> $VMsWithoutTools = Get-VM | Where-Object {($_.ExtensionData.Guest.ToolsStatus -ne
'toolsOk') -and ($_.Name -ne "VCSA")}
```

VM's with VMware Tools up and running can be shutdown cleanly through their GuestOS.

```
PS /> Foreach ($VM in $VMsWithTools) { Stop-VMGuest -Confirm:$false}
```

Guests that cannot accept a shutdown command to the Guest OS can be stopped:

```
PS /> Foreach ($VM in $VMsWithoutTools) { Stop-VM -Confirm:$false}
```

Now to confirm that there are no VM's running other than vCenter:

```
PS /> $RunningVMs = Get-VM | Where-Object {($_.PowerState -eq 'PoweredOn') -and
($_.Name -ne 'VCSA')}

PS /> $RunningVMs.Count
0
```

It is recommended that vCenter is moved to the first host to make it easy to restart it later. To do that we'll need to get a list of hosts in the cluster, sorted by the host, and only select the first one. We'll move vCenter to that host:

```
PS /> $FirstHost = $Cluster | Get-VMHost | Sort-Object Name | Select-Object -First 1
PS /> Get-VM -Name VCSA | Move-VM -Destination $FirstHost

Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
VCSA                 PoweredOn  2        16.000
```

Before going further, let's make certain there are no inaccessible vSAN objects. This isn't exposed natively in PowerCLI today, but Get-VsanView can provide some additional visibility:

```
PS /> $VsHlth = Get-VsanView -Id VsanVcClusterHealthSystem-vsan-cluster-health-system
PS /> $CluMoRef = $Cluster.ExtensionData.MoRef
PS /> $Test = 'objectHealth'
PS /> $View = 'defaultView'
PS /> $ObjHlth =
$VsHlth.VsanQueryVcClusterHealthSummary($CluMoRef,$null,$null,$null,$test,$null,$view)
PS /> $ObjHlth.ObjectHealth.ObjectHealthDetail

NumObjects Health                                                    ObjUuids
---------- ------                                                    --------
        14 healthy                                                   {c8c8485c-943b-
c6b6-ef90-00505682136c, 7aca485c-d413-3c36-df37-005056828cb4, 7eca485c-8e2e-6e20-3f7e-
005056828cb4, cbc8485c-0dab-4a12-9c96-00505682136c...}
         0 nonavailabilityrelatedincompliance
         0 reducedavailabilitywithpolicypendingfailed
         0 reducedavailabilitywithnorebuilddelaytimer
         0 reducedavailabilitywithpolicypending
         0 datamove
         0 inaccessible
         0 nonavailabilityrelatedincompliancewithpolicypendingfailed
         0 reducedavailabilitywithactiverebuild
         0 nonavailabilityrelatedincompliancewithpolicypending
         0 nonavailabilityrelatedreconfig
         0 reducedavailabilitywithnorebuild
```

If there are inaccessible objects, remove them before proceeding.

vCenter could be shutdown at this point, but before shutting vCenter down, it would advantageous to put a list of all the vSAN nodes in an array first.

The Get-VMHost cmdlet will retrieve the vSAN hosts:

```
PS /> $HostList = $Cluster | Get-VMHost | Sort-Object Name
```

With all of the hosts in the $HostList variable, vCenter can be shut down.

```
# Shutdown vCenter
Get-VM -Name “VCSA” | Stop-VMGuest -Confirm:$false}
```

Because vCenter is now down, our VIServer connection to vCenter is no longer available. The $HostList variable contains each of the host, which can be directly connected to.

By directly connecting to each of them, we can put them in maintenance mode and power them down.

```
# Set our host credentials
$User = "root"
$Pass = "VMware1!"

# Loop through each of the hosts in $HostList
Foreach ($VMHost in $HostList.Name) {

    # Connect directly to the current host
    Connect-VIServer -Server $VMHost -user $User -password $Pass

    # Put the current host into maintenance mode with No Action for vSAN data
    Set-VMHost -Server $VMHost -State Maintenance -VsanDataMigrationMode
"NoDataMigration"

    # Wait for the host to enter maintenance mode
    While ((Get-VMHost -Server $VMHost -Name $VMHost).ConnectionState -ne
"Maintenance") {
        Start-Sleep 1
    }

    # When the host has entered maintenance mode, power the host off
    Stop-VMHost -Server $VMHost -Confirm:$false -RunAsync

    # Disconnect from the host
    Disconnect-VIServer -Server $VMHost -Confirm:$false
}
```

Putting all of these parts together, a single script to bring a whole cluster down would look something like this:

```
# Setup variables
$VCSA = ‘VCSA’

# Setup the Cluster Object
$Cluster = Get-Cluster -Name "vSAN"

# Get all of the VM's on the cluster
$PoweredOnVMs = $Cluster | Get-VM | Where-Object {($_.PowerState -eq 'PoweredOn') -and
($_.Name -ne $VCSA)}

# Enumerate the Powered On VM's and power them off - Except for vCenter
Foreach ($VM in $PoweredOnVMs) {
  $Guest = Get-VM -Name $VM
    Write-Host "Shutting down $Guest"
    If ($Guest.ExtensionData.Guest.ToolsStatus -eq 'toolsOk') {
      # If VMware Tools are Ok, shutdown GuestOS
      $Guest | Stop-VMGuest -Confirm:$false
    } else {
      # If VMtools aren’t Ok or not installed Hard Power Off
      $Guest | Stop-VM -Confirm:$false -RunAsync
```

```
    }
}
# Wait until all VM's other than vCenter are powered off
While ((Get-VM | Where-Object {($_.PowerState -eq 'PoweredOn') -and ($_.Name -ne
'VCSA')}).Count -gt "0") {
  Start-Sleep 1
  Write-Host “.” -NoNewLine
}
Write-Host “”
Write-Host "All non-vCenter VM's are powered off"

# Move vCenter to the first host in the cluster alphabetically
$FirstHost = $Cluster | Get-VMHost | Sort-Object Name | Select-Object -First 1
Get-VM -Name VCSA | Move-VM -Destination $FirstHost -RunAsync
Write-Host "Moving vCenter to $FirstHost"

# Ensure the vCenter is on the first host
While ((Get-VM -Name 'VCSA').VMHost.Name -ne $FirstHost.Name) {
    # Wait
    Start-Sleep 1
    Write-Host "." -NoNewline
}
Write-Host ""
Write-Host "vCenter has moved to $FirstHost"

# Use Get-VsanView to determine if there are any objects that have issues.
$VsHlth = Get-VsanView -Id VsanVcClusterHealthSystem-vsan-cluster-health-system
$CluMoRef = $Cluster.ExtensionData.MoRef
$Test = ‘objectHealth’
$View = ‘defaultView’
$ObjHlth =
$VsHlth.VsanQueryVcClusterHealthSummary($CluMoRef,$null,$null,$null,$test,$null,$view)

# Get a sum of all of the non-healthy objects
# Allow the administrator time to resolve objects that aren't healthy
While (($ObjHlth.ObjectHealth.ObjectHealthDetail |Where-Object {$_.Health -ne
"healthy"} |  Measure-Object -Property NumObjects -sum).sum -gt "0") {
          # Wait 10 minutes
          Start-Sleep 600
}
Write-Host "All vSAN Objects are healthy, proceeding"

# Retrieve a list of all vSAN Hosts for the cluster
$HostList = $Cluster | Get-VMHost | Sort-Object Name

# Shutdown vCenter
Get-VM -Name “VCSA” | Stop-VMGuest -Confirm:$false

# Disconnect from vCenter
Disconnect-VIServer -Server $Vcenter -Confirm:$false

# Set our vSAN host credentials
$User = "root"
$Pass = "VMware1!"

# Connect to the first host (where vCenter is running)
Connect-VIServer -Server $FirstHost.Name -user $User -password $Pass
# Check to make certain vCenter is completely powered down
While ( (Get-VM -Name 'VCSA').PowerState -ne 'PoweredOff' ) {
    # Wait
    Start-Sleep 5
    Write-Host "." -NoNewline
}
Write-Host ""
Write-Host "vCenter is now offline"

# Disconnect from the 1st Server
Disconnect-VIServer -Server $FirstHost.Name -Confirm:$false

# Loop through each of the hosts in $HostList
Foreach ($VMHost in $HostList.Name) {
```

```
    # Connect directly to the current host
    Connect-VIServer -Server $VMHost -user $User -password $Pass

    # Put the current host into maintenance mode with No Action for vSAN data
    Set-VMHost -Server $VMHost -State Maintenance -VsanDataMigrationMode
"NoDataMigration"

    # Wait for the host to enter maintenance mode
    While ((Get-VMHost -Server $VMHost -Name $VMHost).ConnectionState -ne
"Maintenance") {
        Start-Sleep 1
    }

    # When the host has entered maintenance mode, power the host off
    Stop-VMHost -Server $VMHost -Confirm:$false -RunAsync

    # Disconnect from the host
    Disconnect-VIServer -Server $VMHost -Confirm:$false
}
```

When all hosts are ready to be powered on, they will need to be taken out of Maintenance Mode. When they have all been taken out of Maintenance Mode, vCenter can be powered on:

```
# Set ESXi Credentials
$User = "root"
$Pass = "VMware1!"

# Manually create our host list
$HostList =
"sc1.scdemo.local","sc2.scdemo.local","sc3.scdemo.local","sc4.scdemo.local","sc5.scdemo.local","sc6.scdemo.local"

# Enumerate each host in the list
Foreach ($VMHost in $HostList) {
  # Connect to the host
  Connect-VIServer -Server $VMHost -user $User -password $Pass
  # Take the host out of maintenance mode
  Set-VMhost $VMHost -Server $VMHost -State Connected
  # Disconnect from the host
  Disconnect-VIServer -Server $VMHost -Confirm:$false
}

# Reconned to the 1st host (where vCenter is)
Connect-VIServer -Server "sc1.scdemo.local" -user root -password VMware1!
# Wait to be certain the host isn't in Maintenance Mode
While ( (Get-VMHost -Name "sc1.scdemo.local").ConnectionState -eq "Maintenance") {
    Start-Sleep 1
    Write-Host "." -NoNewline
}
Write-Host ""
# Start the vCenter VM
Get-VM -Name "VCSA" -Server "sc1.scdemo.local" | Start-VM
# Disconnect from the first host.
Disconnect-VIServer -Server "sc1.scdemo.local" -Confirm:$false
```

Removing Disk Groups from Hosts no longer in a vSAN Cluster
Removing hosts from a vSAN cluster is not a difficult process, but it is important to remember to remove the disk groups from the host before removing it from the cluster.

In the situation where a host has been removed from a vSAN cluster, and one or more disk groups have not been removed, administrators typically remove residual disk groups from the ESXi command line using the ESXCLI command "**esxcli vsan storage remove**".

This requires ssh being enabled for console access, or using the vSphere CLI tool for Windows or Linux to execute the command to remove the disk group's devices.

It can be hard to diagnose when disk groups still exist, as they are not shown in the vSphere Client and are not shown in PowerCLI when using "**Get-VsanDiskGroup -VMHost 'hostname'**". They are shown when using "**esxcli vsan storage list**" from an ESXi shell or using the vSphere CLI.

With esxcli properly showing that the disk group exists on the host, the Get-EsxCli cmdlet can also be used to see if any disk groups exist on a host:

```
# Connect to the esxcli instance on the host
$EsxCli = Get-EsxCli -VMHost "sc9.scdemo.local" -V2

# Get the vSAN disk group devices
$DiskGroupDevices = $EsxCli.vsan.storage.list.invoke()

# Count only the devices that are being used as cache
$DiskGroupCount = ($DiskGroupDevices | Where-Object {$_.IsCapacityTier -ne
$true}).Count

# Report any disk groups
If ($DiskGroupCount -gt 0) {
    Write-Host "Disk Group(s):"$DiskGroupCount
} else {
    Write-Host "No Disk Groups on the host"
}
```

If there are disk groups, Get-EsxCli can be used to remove the disk groups without having to use SSH or the vSphere CLI:

```
# Connect to the esxcli instance on the host
$EsxCli = Get-EsxCli -VMHost "sc9.scdemo.local" -V2

# Retrieve the disk group devices
$DiskGroupDevices = $EsxCli.vsan.storage.list.invoke()

# Enumerate all of the disk group devices
ForEach ($DiskGroupDevice in $DiskGroupDevices) {

    # Since capacity devices cannot be part of a disk group without a cache device
    # If the device is cache device, then remove it to remove the disk group
    If ($DiskGroupDevice.IsCapacityTier -ne $true) {

        # Notify that the cache device will be removed
        Write-Host "Removing Cache Device:" $DiskGroupDevice.Device

        # Set the device as an argument for removal
        $Args = $esxcli.vsan.storage.remove.CreateArgs()
        $Args.ssd = $DiskGroupDevice.Device

        # Remove the cache device, removing the disk group
        $EsxCli.vsan.storage.remove.Invoke($Args)
    }
}
```

Moving VMs off of a vSAN Host without DRS
vSAN licensing is separate from vSphere licensing. It is common to see vSAN used with versions of vSphere that do not include vSphere DRS as a service on the vSAN Cluster.

Here is an example script that will find the VM's on a vSphere host, move them to another host, and then put the host in Maintenance Mode:

```
# Retrieve the cluster object
$Cluster = Get-Cluster -Name "vSAN"

# Retrieve any connected hosts in the cluster
$VMHosts = $Cluster | Get-VMHost | Where-Object {$_.State -eq "Connected"}

# Host to be put in Maintenance Mode
$MMHost = Get-VMHost -Name "sc1.scdemo.local"

# vMotion all of the VM's from the current host to random partners in the cluster
$VMS = Get-VMHost $MMHost | Get-VM
Foreach ($VM in $VMS) {
          $DestHost = $VMHosts | Get-Random
          Write-Output "Moving VM $VM to $DestHost"
          Move-VM -VM $vm -Destination $dsthost -Confirm:$false | Out-Null
}

Set-VMHost -VMHost $MMhost -State "Maintenance" -VsanDataMigrationMode
"EnsureAccessibility" -Confirm:$false
```

This example was taken from Damian Karlson's post here: https://damiankarlson.com/2010/10/13/cluster-evacuation-reboot-without-drs-powercli/

vSAN Storage Policies

vSAN Storage Policies are used to determine data placement and availability of objects on a vSAN datastore.

Storage Policies, while used by vSAN objects, are actually part of the Storage Policy Based Management framework. There are dedicated PowerCLI cmdlets that allow for retrieving the SPBM supported features of a datastore, retrieving and setting the current policy for a given vSAN object, and more.

Storage Polices are comprised of rules that are applied to VM or its components at the host or datastore layer. This section is going to focus specifically on Storage Polices as they relate to vSAN datastores.

Creating new vSAN Storage Polices

When vSAN is configured on a vSphere cluster, a default Storage Policy is applied to all objects on a vSAN datastore.

Different workloads can often require different storage policies, or it could be desirable to have a different storage policy. It really depends on the workload, or the desire of the administrator.

To create a Storage Policy, it is important to understand the makeup of a Storage Policy.

A Storage Policy is going to require a couple obvious properties, like name and description, but will also require one or more rules, or array of rules, called a ruleset.

Common Rules apply to the VMware API for I/O Filtering (VAIO) namespace. A couple examples of these include VM Encryption and

Storage I/O Control. We're not going to focus on Common Rules, but rather AnyOfRulesets as they pertain to vSAN.

When creating a vSAN Storage Policy, different vSAN Rules are selected. The default vSAN Storage Policy is created with the following rules:

1. Primary Failures To Tolerate = 1
2. Failure Protection Method = Mirroring

To create this policy in PowerCLI, the New-SpbmStoragePolicy cmdlet is used:

```
PS /> New-SpbmStoragePolicy -Name "RAID1 Mirroring" -Description "Mirroring Policy" -
AnyOfRuleSets (New-SpbmRuleSet(New-SpbmRule -Capability (Get-SpbmCapability -Name
"VSAN.hostFailuresToTolerate") -Value 1),
(New-SpbmRule -Capability (Get-SpbmCapability -Name "VSAN.replicaPreference") -Value
"RAID-1 (Mirroring) - Performance"))

Name                 Description                          Rule Sets
Common Rules
----                 -----------                          ---------
------------
RAID1 Mirroring      Mirroring Policy
{(VSAN.hostFailuresToTolera... {}
```

That is a bit lengthy and difficult to follow, so let's break it down a bit. To do that, let's first set our vSAN rules to their own variables:

```
PS /> Get-SpbmCapability -Name "VSAN*"

Name                          ValueType         AllowedValue
----                          ---------         ------------
VSAN.cacheReservation         System.Int32      0 .. 1000000
VSAN.checksumDisabled         System.Boolean    True .. False
VSAN.forceProvisioning        System.Boolean    True .. False
VSAN.hostFailuresToTolerate   System.Int32      0 .. 3
VSAN.iopsLimit                System.Int32      0 .. 2147483647
VSAN.locality                 System.String     {None, Preferred Fault Domain, Second...
VSAN.proportionalCapacity     System.Int32      0 .. 100
VSAN.replicaPreference        System.String     {RAID-1 (Mirroring) - Performance, RA...
VSAN.stripeWidth              System.Int32      1 .. 12
VSAN.subFailuresToTolerate    System.Int32      0 .. 3
```

Now let's assign each of these capability objects to their own readable variable, where they can more easily be reused:

```
PS /> $VsanPFTT           = Get-SpbmCapability -Name "VSAN.hostFailuresToTolerate"
$VsanFTM            = Get-SpbmCapability -Name "VSAN.replicaPreference"
$VsanSFTT           = Get-SpbmCapability -Name "VSAN.subFailuresToTolerate"
$VsanOSR            = Get-SpbmCapability -Name "VSAN.proportionalCapacity"
$VsanCacheRes       = Get-SpbmCapability -Name "VSAN.cacheReservation"
$VsanCheckSumOff    = Get-SpbmCapability -Name "VSAN.checksumDisabled"
$VsanIopsLimit      = Get-SpbmCapability -Name "VSAN.iopslimit"
$VsanLocality       = Get-SpbmCapability -Name "VSAN.locality"
$VsanStripeWidth    = Get-SpbmCapability -Name "VSAN.stripeWidth"
$VsanForceProvision = Get-SpbmCapability -Name "VSAN.forceProvisioning"

$FTM = Get-SpbmCapability -Name "VSAN.replicaPreference" | Select-Object AllowedValue
$Locality = Get-SpbmCapability -Name "VSAN.Locality" | Select-Object AllowedValue
```

Notice that the possible values for Failure Tolerance Method (FTM) are

being placed into the $FTM variable. These are text strings that could be misspelled, and it easier to add them to an array, simply calling the array item. The same is done above for Locality.

Now that those variables are setup, the above creation of a Mirroring policy can be followed more easily. **Backticks tell PowerShell to continue the next line as it is part of the same instruction.*

```
# New Mirroring Policy
New-SpbmStoragePolicy -Name "RAID1 Mirroring" -Description "Mirroring Policy" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 1), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[0]) `
    )
```

Creating a RAID5 policy is easy as well:

```
# New RAID5 Policy
New-SpbmStoragePolicy -Name "RAID5" -Description "RAID5 Erasure Coding" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 1), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[1]) `
    )
```

And creating a RAID6 policy:

```
# New RAID6 Policy
New-SpbmStoragePolicy -Name "RAID5" -Description "RAID6 Erasure Coding" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 2), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[1]) `
    )
```

So far, the basic rules have been addressed. But what if an administrator wants to make a vSAN policy that has the following rules?

- PFTT=1
- FTM=Mirroring
- Object Space Reservation = 75%
- IOPS Limit = 5000

Expanding the above examples, we can add OSR and IOPS limits. Consider though that OSR & IOPS limits are Integer values.

```
# New Custom Policy
New-SpbmStoragePolicy -Name "Mirrored-75-SpaceReserved-5K-Limit" `
    -Description "Mirroring with 75% Space Reservation & 5K IOPS" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 1), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[0]), `
    (New-SpbmRule -Capability $VsanOSR -Value ([int]‘75’)), `
    (New-SpbmRule -Capability $VsanIopsLimit -Value ([int]‘5000’))
    )
```

A script that could create several policies at once would look something like this:

```
# Create vSAN Storage Policy Variables
$VsanPFTT           = Get-SpbmCapability -Name "VSAN.hostFailuresToTolerate"
$VsanFTM            = Get-SpbmCapability -Name "VSAN.replicaPreference"
$VsanSFTT           = Get-SpbmCapability -Name "VSAN.subFailuresToTolerate"
$VsanOSR            = Get-SpbmCapability -Name "VSAN.proportionalCapacity"
$VsanCacheRes       = Get-SpbmCapability -Name "VSAN.cacheReservation"
$VsanCheckSumOff    = Get-SpbmCapability -Name "VSAN.checksumDisabled"
$VsanIopsLimit      = Get-SpbmCapability -Name "VSAN.iopslimit"
$VsanLocality       = Get-SpbmCapability -Name "VSAN.locality"
$VsanStripeWidth    = Get-SpbmCapability -Name "VSAN.stripeWidth"
$VsanForceProvision = Get-SpbmCapability -Name "VSAN.forceProvisioning"

$FTM = Get-SpbmCapability -Name "VSAN.replicaPreference" | Select-Object AllowedValue
$Locality = Get-SpbmCapability -Name "VSAN.Locality" | Select-Object AllowedValue

# Create policies for vSAN
Write-Host “Creating Policies for vSAN”
Write-Host “---------------------------------”

# New RAID1 Mirroring Policy
Write-Host “Creating a RAID1 Mirroring Policy”
New-SpbmStoragePolicy -Name "RAID1 Mirroring" -Description "Mirroring Policy" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 1), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[0]) `
    )

# New RAID5 Policy
Write-Host “Creating a RAID5 Policy”
New-SpbmStoragePolicy -Name "RAID5" -Description "RAID5 Erasure Coding" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 1), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[1]) `
    )

# New RAID6 Policy
Write-Host “Creating a RAID5 Policy”
New-SpbmStoragePolicy -Name "RAID5" -Description "RAID5 Erasure Coding" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 2), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[1]) `
    )

# New Custom Policy
Write-Host “Creating a Custom Policy”
New-SpbmStoragePolicy -Name "Mirrored-75-SpaceReserved-5K-Limit" `
    -Description "Mirroring with 75% Space Reservation & 5K IOPS" `
    -AnyOfRuleSets (New-SpbmRuleSet `
    (New-SpbmRule -Capability $VsanPFTT -Value 1), `
    (New-SpbmRule -Capability $VsanFTM -Value $FTM.AllowedValue[0]), `
    (New-SpbmRule -Capability $VsanOSR -Value ([int]‘75’)), `
    (New-SpbmRule -Capability $VsanIopsLimit -Value ([int]‘5000’))
    )
```

Backing up vSAN Storage Policies

PowerCLI can make the process of creating Storage Policies relatively easy.

For environments that have tens or hundreds of Storage Policies, it is as important to be able to back these policies up.

The Export-SpbmStoragePolicy cmdlet can be used to export Storage Policies to a specific path on the host executing the cmdlet. The location (FilePath) and the Storage Policy Object are required.

Backing up the RAID1 Mirroring Storage Policy above:

```
# Backup RAID1 Mirroring Policy

# Put the Policy Object into $Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID1 Mirroring”

# Create a path for the Policy
$FilePath = “/Users/Admin/SPBM/”+$Policy.Name+”.xml”
# Remove any spaces in the path
$FilePath = $FilePath -Replace (‘ ‘)

# Export (backup) the policy
Export-SpbmStoragePolicy -StoragePolicy $Policy -FilePath -Path
```

To back up all the Storage Policies on a vCenter Server, it is only required to enumerate all of the policies and export each of them:

```
# Back up all storage policies

# Get all of the Storage Policies
$StoragePolicies = Get-SpbmStoragePolicy

# Loop through all of the Storage Policies
Foreach ($Policy in $StoragePolicies) {
    # Create a path for the current Policy
    $FilePath = "/Users/Admin/SPBM/"+$Policy.Name+".xml"
    # Remove any spaces from the path
    $FilePath = $FilePath -Replace (' ')
    # Export (backup) the policy
    Export-SpbmStoragePolicy -StoragePolicy $Policy -FilePath $FilePath
}
```

Restoring vSAN Storage Policies

Just as important to backing up vSAN Storage Policies, is restoring those policies if necessary.

Rather than using the Export-SpbmStoragePolicy cmdlet, we’ll use the Import-SpbmStoragePolicy cmdlet instead.

```
# Recover the RAID 1 Mirroring Policy

# Get the RAID1 Policy XML file
$PolicyFile = Get-Item “/Users/Admin/SPBM/RAID1.xml”

# Import the policy
Import-SpbmStoragePolicy -Name “RAID1” -Description “RAID1” -FilePath $PolicyFile
```

Having to manually enter the Name & Description values make this a little difficult to scale.

Reading the contents of the XML and automatically placing this information in these fields would allow for only the file name and path to be passed:

```
# Recover the RAID 1 Mirroring Policy

# Get the RAID1 Policy XML file
$PolicyFile = Get-Item “/Users/Admin/SPBM/RAID1.xml”

# Read the contents of the policy file to set variables
$PolicyFileContents = [xml](Get-Content $PolicyFile)
```

```
# Get the Policy's name & description
$PolicyName = $PolicyFileContents.PbmCapabilityProfile.Name.'#text'
$PolicyDesc = $PolicyFileContents.PbmCapabilityProfile.Description.'#text'

# Import the policy
Import-SpbmStoragePolicy -Name $PolicyName -Description $PolicyDesc -FilePath
$PolicyFile
```

Pretty easy! Import all the policy xml files that reside in a single directory easily as well:

```
# Recover the Policies in /Users/Admin/SPBM/
$PolicyFiles = Get-ChildItem "/Users/Admin/SPBM/" -Filter *.xml

# Enumerate each policy file found
Foreach ($PolicyFile in $PolicyFiles) {

  # Get the Policy XML file path
  $PolicyFilePath = $PolicyFile.FullName

  # Read the contents of the policy file to set variables
  $PolicyFileContents = [xml](Get-Content $PolicyFilePath)

  # Get the Policy's name & description
  $PolicyName = $PolicyFileContents.PbmCapabilityProfile.Name.'#text'
  $PolicyDesc = $PolicyFileContents.PbmCapabilityProfile.Description.'#text'

  # Import the policy
  Import-SpbmStoragePolicy -Name $PolicyName -Description $PolicyDesc -FilePath
$PolicyFile
}
```

With policies recovered, they can be applied to vSAN objects.

Applying vSAN Storage Policies to a VM or its Drives

Whether Storage Polices are newly created, recently restored, or already existing in a vCenter server, they can then be used for virtual machines and their drives.

Applying a Storage Policy to VM on vSAN can be performed using the Set-SpbmEntityConfiguration cmdlet.

```
# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name "RAID5"

# Get the VM to apply the policy to
$VM = $Cluster| Get-VM -Name "APP1"

# Get the current SPBM config and replace the policy assigned to it
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration $VM) -
StoragePolicy $Policy
```

This applies the Storage Policy to the VM's Namespace, not the hard disks that the VM contains. In vSAN 6.7, where the Swap file inherits the Storage Policy of the Namespace, the Storage Policy will be applied to the Swap file as well.

To apply the Storage Policy to the VM and its hard disks, we'll have to retrieve the disks attached to the VM and apply the policy to each of them.

```
# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID5”

# Get the VM to apply the policy to
$VM = $Cluster| Get-VM -Name “APP1”

# Get the current SPBM configuration & replace the policy assigned
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration $VM) -
StoragePolicy $Policy

# Get the current SPBM configuration for each hard disk & replace the policy assigned
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration -HardDisk (Get-
HardDisk -VM $VM)) -StoragePolicy $Policy
```

But what if we only want to apply a Storage Policy to a single virtual disk that belongs to the VM? That is easy as well:

```
# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID1”

# Get the VM to apply the policy to
$VM = $CLUSTER | Get-VM -Name “SQL1”

# Get the VM’s Drive that the policy will be assigned to (say one that is 60GB)
$HD = $Cluster | Get-HardDisk -VM $VM | Where-Object {$_.CapacityGB -eq “60”}

# Get the current SPBM config and replace the policy assigned to it
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration $HD) -
StoragePolicy $Policy
```

What about a situation where we want to apply a policy to multiple VM’s simultaneously?

```
# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID5”

# Get the VM’s the policy is going to be applied to
$AppVMs = $Cluster | Get-VM | Where-Object {$_.Name -like “APP*”}

# Get the each AppVM SPBM config & replace the policy assigned
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration | $AppVMs) -
StoragePolicy $Policy

# Get the current SPBM configuration for each vmdk & replace the policy assigned
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration -HardDisk (Get-
HardDisk -VM $AppVMs)) -StoragePolicy $Policy
```

This iteration of the script will get a list of VMs, change the Storage Policy assigned to each of the VM’s & Namespaces, and then do the same for each hard disk.

This isn’t the “cleanest” way of doing this, as it is probably more desirable to configure a VM, its Namespace, and then its disks before proceeding to the next VM.

To do that, a Foreach look is a better approach.

```
# Get the working Cluster
$Cluster = Get-Cluster -Name "vSAN"

# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID5”

# Get the VM to apply the policy to
$AppVMs = $Cluster = Get-VM

# Loop through the different VMs
Foreach ($VM in $AppVMs) {
    # Display which VM we're applying the Storage Policy to
    Write-Host "Applying $Policy to $VM"

    # Get the current SPBM configuration & replace the policy assigned
    Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration $VM) -
StoragePolicy $Policy

    # Get the current SPBM configuration for each vmdk & replace the policy assigned
    Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration -HardDisk
(Get-HardDisk -VM $VM)) -StoragePolicy $Policy
}
```

Remember that applying a policy to a VM could potential require a full copy of the VM, depending on the policy rules.

Applying a policy to multiple VMs simultaneously could potentially require a significant amount additional capacity even if only temporarily.

Consider the situation where a 4-node cluster has 600 VMs, where each of those VM’s has an Object Space Reservation rule, and the cluster is at 80% capacity utilization. If an administrator were to apply a new policy (or even change an existing policy), and that policy required a full copy to be made (like moving from Mirroring to RAID5), the cluster would run out of capacity before being able to change the policy on each VM.

It is far better to set Storage Policy configurations in batches of VMs to prevent this issue.

```
# Get the working Cluster
$Cluster = Get-Cluster -Name "vSAN"

# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID5”

# Get the VM’s the policy is going to be applied to
$AppVMs = $Cluster | Get-VM

# Group VMs into groups of 10
$Counter = [pscustomobject] @{ Value = 0}
$VmGroups = $AppVMs | Group-Object -Property { [math]::Floor($Counter.Value++/10)}

# Loop through the number of groups we have
For ($i=0; $i -le $VmGroups.Count-1; $i++) {
  # Write the Current working Group
  Write-Host "Group $i"

  # Enumerate each VM in the current Group
  Foreach ($GuestVM in $VmGroups[$i].Group) {
    # Apply the policy to the current VM in the Group
    Write-Host "Applying $Policy for $GuestVM"
    # Set the Policy for the VM Namespace
```

```
     Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration -VM
$GuestVM) -StoragePolicy $Policy
     # Set the Policy for any Hard Disks
     Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration -HardDisk
(Get-HardDisk -VM $GuestVM)) -StoragePolicy $Policy
  }
  While ((Get-VsanResyncingComponent -Cluster $Cluster)) {
  Write-Host "." -ForegroundColor "DarkYellow" -NoNewline
  }
}
```

**Many thanks to Timo Sugliani for bringing attention to an issue with the logic in this section in releases 1.0-1.2 of this document.*

Changing the Storage Policy for All Objects with a Given Policy

Setting Storage Policies for vSAN Objects that meet a naming scheme is one thing. Updating Storage Policies for vSAN Objects with a given Storage Policy is another.

In the previous section we covered setting a Storage Policy variable, and applying that policy to a given entity, such as a VM or hard disk.

```
# Get the working SPBM Policy
$Policy = Get-SpbmStoragePolicy -Name “RAID5”

# Get the VM to apply the policy to
$VM = $Cluster| Get-VM -Name “APP1”

# Get the current SPBM config and replace the policy assigned to it
Set-SpbmEntityConfiguration -Configuration (Get-SpbmEntityConfiguration $VM) -
StoragePolicy $Policy
```

But what if we have a policy that no longer has the ability to be compliant on a vSAN datastore?

```
PS /> Get-SpbmEntityConfiguration

Entity                Storage Policy           Status          Time Of Check
------                --------------           ------          -------------
Hard disk 1           RAID5                    nonCompliant    2/7/19 12:36:16 AM
Hard disk 3           RAID5                    nonCompliant    2/7/19 12:36:16 AM
SC1                   RAID5                    nonCompliant    2/7/19 12:36:16 AM
SC6                   RAID5                    nonCompliant    2/7/19 12:36:16 AM
WITNESS2              RAID5                    nonCompliant    2/7/19 12:36:16 AM
Hard disk 1           RAID5                    nonCompliant    2/7/19 12:36:16 AM
H5CLIENT              RAID5                    nonCompliant    2/7/19 12:36:16 AM
SC1                   RAID5                    nonCompliant    2/7/19 12:36:16 AM
SC5                   RAID5                    nonCompliant    2/7/19 12:36:16 AM
Hard disk 3           RAID5                    nonCompliant    2/7/19 12:36:16 AM
SC3                   RAID5                    nonCompliant    2/7/19 12:36:16 AM
vRouter0              RAID5                    nonCompliant    2/7/19 12:36:16 AM
SC4                   RAID5                    nonCompliant    2/7/19 12:36:16 AM
Hard disk 2           RAID5                    nonCompliant    2/7/19 12:36:16 AM
```

The output above is the result of permanently removing the fourth host of a four host vSAN Cluster. Remember that a RAID5 (Erasure Coding) Storage Policy ***requires a minimum of four nodes***.

That’s correct, while working on something else, the author removed a host, and completely forgot that a RAID5 Storage Policy was in place.

To recover from this using the UI, every non-compliant object would have to have a new Storage Policy assigned.

There is no way to make a vSAN Object with a RAID5 Storage Policy compliant with only three nodes.

If the cluster has hundreds, or thousands of objects, this could be very time consuming.

Using the previous batch script to apply a Storage Policy, some modifications can be made. For starters, let's only retrieve the entities that had the old policy.

```
# Get the working Cluster
$Cluster = Get-Cluster -Name "vSAN"

# Get the old SPBM Policy
$OldPolicy = Get-SpbmStoragePolicy -Name "RAID5"

# Get the new SPBM Policy
$NewPolicy = Get-SpbmStoragePolicy -Name “vSAN Default Storage Policy”

# Get the vSAN objects with the now-defunct policy so we can apply the new policy
$VsanObjectsOldPolicy = Get-SpbmEntityConfiguration -StoragePolicy $OldPolicy
```

With $VsanObjectsOldPolicy containing the entities that have the RAID5 Storage Policy (now defunct and noncompliant), we can set a new, the default vSAN Policy in this case, Storage Policy to them, so they can become compliant again.

```
# Group VMs into groups of 3
$Counter = [pscustomobject] @{ Value = 0}
$ObjectGroups = $VsanObjectsOldPolicy.Entity | Group-Object -Property
{[math]::Floor($Counter.Value++/3)}

# Loop through the number of groups we have
For ($i=0; $i -le $ObjectGroups.Count-1; $i++) {
  # Write the Current working Group
  Write-Host "Group $i"

  # Enumerate each Entity in the current Group
  Foreach ($OldObject in $ObjectGroups[$i].Group) {
    # Apply the policy to the current Entity in the Group
    Write-Host "Updating Storage Policy for $OldObject"

    Set-SpbmEntityConfiguration -Configuration $OldObject -StoragePolicy $NewPolicy
  }
  Write-Host "Waiting for any vSAN Object Resyncs to complete" -ForegroundColor
"DarkYellow"
  While ((Get-VsanResyncingComponent -Cluster $Cluster)) {
    Write-Host "." -ForegroundColor "DarkYellow" -NoNewline
  }
  Write-Host "* " -ForegroundColor "Green"
  Write-Host "No vSAN Resyncs Pending" -ForegroundColor "Green"
}
```

Putting both of these together, we see:

```
# Get the working Cluster
$Cluster = Get-Cluster -Name "vSAN"

# Get the old SPBM Policy
$OldPolicy = Get-SpbmStoragePolicy -Name "RAID5"
```

```
# Get the new SPBM Policy
$NewPolicy = Get-SpbmStoragePolicy -Name “vSAN Default Storage Policy”

# Get the vSAN objects with the now-defunct policy so we can apply the new policy
$VsanObjectsOldPolicy = Get-SpbmEntityConfiguration -StoragePolicy $OldPolicy

# Group VMs into groups of 3
$Counter = [pscustomobject] @{ Value = 0}
$ObjectGroups = $VsanObjectsOldPolicy.Entity | Group-Object -Property
{[math]::Floor($Counter.Value++/3)}

# Loop through the number of groups we have
For ($i=0; $i -le $ObjectGroups.Count-1; $i++) {
  # Write the Current working Group
  Write-Host "Group $i"

  # Enumerate each Entity in the current Group
  Foreach ($OldObject in $ObjectGroups[$i].Group) {
    # Apply the policy to the current Entity in the Group
    Write-Host "Updating Storage Policy for $OldObject"

    Set-SpbmEntityConfiguration -Configuration $OldObject -StoragePolicy $NewPolicy
  }
  Write-Host "Waiting for any vSAN Object Resyncs to complete" -ForegroundColor
"DarkYellow"
  While ((Get-VsanResyncingComponent -Cluster $Cluster)) {
    Write-Host "." -ForegroundColor "DarkYellow" -NoNewline
  }
  Write-Host "* " -ForegroundColor "Green"
  Write-Host "No vSAN Resyncs Pending" -ForegroundColor "Green"
}
```

The resulting output looks something like this:

```
Group 0
Updating Storage Policy for SC5

TimeOfCheck      : 2/7/19 1:18:50 AM
ComplianceStatus : compliant
Entity           : SC5
StoragePolicy    : vSAN Default Storage Policy
ReplicationGroup :
Name             : SC5
Id               : VirtualMachine-vm-215

Updating Storage Policy for SC1

TimeOfCheck      : 2/7/19 1:18:54 AM
ComplianceStatus : compliant
Entity           : SC1
StoragePolicy    : vSAN Default Storage Policy
ReplicationGroup :
Name             : SC1
Id               : VirtualMachine-vm-212

Updating Storage Policy for Hard disk 1

TimeOfCheck      : 2/7/19 1:18:57 AM
ComplianceStatus : nonCompliant
Entity           : Hard disk 1
StoragePolicy    : vSAN Default Storage Policy
ReplicationGroup :
Name             : Hard disk 1
Id               : VirtualMachine-vm-352/2000
...Group 1
```

Individually setting the Storage Policy on vSAN Objects is relatively easy in the UI. Setting the policy for several, multiple, hundreds, or

even thousands of objects is a bit more challenging when performed manually.

This is another example of how PowerCLI can be used to more efficiently manage an environment at scale, preventing user error and maintaining consistency.

vSAN Stretched Cluster Operations

With the introduction of Stretched Clusters to vSAN in version 6.1, it has been even easier for customers to take advantage of Active-Active availability across sites for vSAN workloads.

Stretched Cluster vSAN configurations behave a little bit differently than traditional vSAN clusters in that they have two fault domains and require a vSAN Witness Host.

A few of the nuances of Stretched Cluster configuration have been covered in the Configuration Recipes section, such as how to set static routes for vSAN VMkernel interfaces, as well how to configure Witness Traffic Separation for configurations that support it.

Some operational tasks associated with vSAN Stretched Clusters that are a bit different than traditional vSAN clusters. Some of the tasks include patching, selecting the alternate site to be preferred, swapping a vSAN Witness Host, or configuring virtual machines to reside on one fault domain or the other.

Changing the "Preferred" Site

Fault domains in stretched vSAN cluster configurations are comprised of a "Preferred" fault domain and a "Non-Preferred" fault domain. Typically, one fault domain will be one site and the alternate fault domain will the alternate site.

The preferred designation is directly related to how each site will behave in the event of a site isolation. More information around the preferred fault domain and vSAN Stretched Cluster failure scenarios can be found on https://storagehub.vmware.com/ in the Stretched Cluster guide.

The two different fault domains can have any name. The name does not have to align with the current designation (preferred or not).

Sometimes a virtualization administrator may wish to select the alternate site as the preferred. Cases where this may be desirable are scenarios where a site is being taken offline for a given amount of time, or possibly during an upgrade process.

The Get-VsanFaultDomain will return fault domain objects for use by PowerCLI.

```
PS /> Get-VsanFaultDomain -Cluster $Cluster

Name                           Cluster
----                           -------
Preferred                      vSAN
Secondary                      vSAN
```

Fault domains can be retrieved and stored in a variable using their name:

```
PS /> $PreferredFd = Get-VsanFaultDomain -Cluster $Cluster -Name "Preferred"
PS /> $SecondaryFd = Get-VsanFaultDomain -Cluster $Cluster -Name "Secondary"
```

Also, consider that Stretched Cluster vSAN configurations only have two fault domains, with one of them being designated as “Preferred.”

The “PreferredFaultDomain” property returned by the Get-VsanClusterConfiguration cmdlet contains the name of the fault domain with the Preferred designation.

Query all fault domains in a stretched cluster (remember there are 2), and only retrieving the one that matches the name of the .PreferredFaultDomain property, makes it easy to dynamically return one of the two fault domains.

```
PS /> $PreferredFd = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -eq
(Get-VsanClusterConfiguration -Cluster $Cluster).PreferredFaultdomain}
PS />
```

Performing the same query, but selecting only the fault domain that **does not** contain the name of the fault domain, makes it easy to dynamically return the alternate (or non-preferred) fault domain.

```
PS /> $SecondaryFd = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -ne
(Get-VsanClusterConfiguration -Cluster $Cluster).PreferredFaultdomain}
PS />
```

With the Preferred fault domain and Secondary fault domain set to variables, swapping the preferred fault domain from Preferred to Secondary is accomplished with the Set-VsanClusterConfiguration cmdlet.

```
PS /> Get-VsanClusterConfiguration -Cluster $Cluster | Set-VsanClusterConfiguration -
PreferredFaultDomain $SecondaryFd
```

After this is performed, $SeconaryFd becomes the “Preferred” fault domain.

To change it back the $PreferredFD variable that was previously created would have to be used to set the PreferredFaultDomain parameter.

```
PS /> Get-VsanClusterConfiguration -Cluster $Cluster | Set-VsanClusterConfiguration -
PreferredFaultDomain $PreferredFd
```

Alternatively, if we simply wanted to toggle from whichever fault domain is the current preferred fault domain, we could get a bit more elaborate:

```
PS /> Get-VsanClusterConfiguration -Cluster $Cluster  | Set-VsanClusterConfiguration -
PreferredFaultDomain (Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -ne
(Get-VsanClusterConfiguration -Cluster $Cluster).PreferredFaultdomain})
```

What exactly are we doing here?

- Retrive the Cluster configuration (Get-VsanClusterConfiguration)
- Take the configuration and pipe it into the next set of steps'
- Retrieve the fault domain that isn't currently the preferred
- Set that fault domain to the preferred

This logic simply toggles the currently non-preferred fault domain to the preferred fault domain.

By dynamically picking fault domains there is no need to worry about incorrectly specifying the different fault domains as they are assigned to variables. Using those dynamically generated fault domain variables, we could check against other logic and choose to (or not to) change the preferred fault domain.

If we simply want to swap, regardless of which one is currently preferred, that is easy as well.

Patching a vSAN Stretched Cluster

Patching any vSphere cluster is easily accomplished using Update Manager. In the samples mentioned in the Patching section, an update operation is executed against the entire cluster.

This will work for stretched vSAN clusters as well, but the general guidance from VMware is to patch each fault domain separately in an ordered fashion.

*Remember that Update Manager PowerCLI cmdlets are only available in Windows using PowerShell, and not PowerShell Core. The next pieces of code should be executed from a Windows system with PowerShell and PowerCLI.

Update Manager does not distinguish which hosts are in which site as vSAN fault domains do, so the examples collecting the vSAN fault domains and putting them into a variable are of significant value here.

Rather that depending on Update Manager to handle the host selection process, PowerCLI can be used to select hosts in an individual site for updating. Once one site has been updated, the other site can be updated. This will be followed by the vSAN Witness Host, which is recommended to be updated last.

For the PreferredFD:

```
PS C:\> $PreferredFd = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -
eq ((Get-VsanClusterConfiguration -Cluster $Cluster).PreferredFaultDomain)}
```

For the NonPreferredFD:

```
PS C:\> $NonPreferredFd = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name
-ne ((Get-VsanClusterConfiguration -Cluster $Cluster).PreferredFaultDomain)}
```

With fault domains designed, hosts in each can be updated. Just to check our logic, let's just enumerate the hosts in each fault domain.

```
PS C:\> $Cluster = Get-Cluster -Name "vSAN"
$ClusterConfiguration = Get-VsanClusterConfiguration -Cluster $Cluster
$PreferredFd = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -eq
$ClusterConfiguration.PreferredFaultdomain}
$SecondaryFD = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -ne
$ClusterConfiguration.PreferredFaultdomain}

Write-Host $PreferredFd
Foreach ($VMHost in ($PreferredFd | Get-VMHost)) {
    Write-Host $VMHost
}
Write-Host $SecondaryFD
Foreach ($VMHost in ($SecondaryFd | Get-VMHost)) {
    Write-Host $VMHost
}

Preferred
sc1.scdemo.local
sc3.scdemo.local
sc2.scdemo.local
Secondary
sc6.scdemo.local
sc5.scdemo.local
sc4.scdemo.local
PS C:\>
```

Rolling through each fault domain, we can put each host in Maintenance Mode, update it with Update Manager, wait until it comes back online, take it out of Maintenance Mode, and move to the next host in the fault domain. When one fault domain (site) is complete, we can move to the alternate site. *Assuming DRS is FullyAutomated

The process to update an individual host is basically:

- Check to see if the host is missing any patches
- If the host is missing patches, put the host in Maintenance Mode
- Patch the host with any missing patches
- Take the host out of Maintenance Mode
- A snippet to do this on a single host would look like this:

```
$TestHost = Get-VMHost -Name "sc10.scdemo.local"
Test-Compliance -Entity $TestHost

$NonCompBase = Get-Compliance -Entity $TestHost | Where-Object {$_.Status -ne
"Compliant"}

    Write-Host $TestHost "not compliant with baseline: " $NonCompBase.Baseline.Name
    Foreach ($NonComp in $NonCompBase.BaseLine) {
        Write-Host "Patching $TestHost with baseline:" $NonComp.Name
```

```
        $TestHost | Update-Entity -Baseline (Get-Baseline -Name $NonComp.Name) -
Confirm:$False}
```

Combining the snippets would look something like this:

```
# Get the Cluster Object
$Cluster = Get-Cluster -Name "vSAN"
# Get the vSAN Configuration
$ClusterConfiguration = Get-VsanClusterConfiguration -Cluster $Cluster
# Get the Preferred Fault Domain
$PreferredFd = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -eq
$ClusterConfiguration.PreferredFaultdomain}
# Get the NonPreferred Fault Domain
$SecondaryFD = Get-VsanFaultDomain -Cluster $Cluster | Where-Object {$_.Name -ne
$ClusterConfiguration.PreferredFaultdomain}

# Write that we're working on the Preferred FD
Write-Host "Updating $PreferredFd"
Foreach ($VMHost in ($PreferredFd | Get-VMHost)) {
  # Check the patch compliance of the current host
  Test-Compliance -Entity $VMHost
  # Get a list of non-compliant baselines
  $NonCompBase = Get-Compliance -Entity $VMHost | Where-Object {$_.Status -ne
"Compliant"}

    # Notify that we're not compliant with X baseline(s)
    Write-Host $VMHost "not compliant with baseline: " $NonCompBase.Baseline.Name

    # Enumerate each baseline we're not compliant with and patch the host
    Foreach ($NonComp in $NonCompBase.BaseLine) {
      # Report which baseline is the host is being patched with
      Write-Host "Patching $TestHost with baseline:" $NonComp.Name
      $VMHost | Update-Entity -Baseline (Get-Baseline -Name $NonComp.Name) -
Confirm:$False
    }
}
Write-Host $SecondaryFD
Foreach ($VMHost in ($SecondaryFd | Get-VMHost)) {
  # Check the patch compliance of the current host
  Test-Compliance -Entity $VMHost
  # Get a list of non-compliant baselines
  $NonCompBase = Get-Compliance -Entity $VMHost | Where-Object {$_.Status -ne
"Compliant"}

    # Notify that we're not compliant with X baseline(s)
    Write-Host $VMHost "not compliant with baseline: " $NonCompBase.Baseline.Name

    # Enumerate each baseline we're not compliant with and patch the host
    Foreach ($NonComp in $NonCompBase.BaseLine) {
      # Report which baseline the host is being patched with
      Write-Host "Patching $TestHost with baseline:" $NonComp.Name
      $VMHost | Update-Entity -Baseline (Get-Baseline -Name $NonComp.Name) -
Confirm:$False
    }
}
```

- When the hosts in the Preferred and Secondary fault domains have been patched, the vSAN Witness Host will need to be patched. To do that, we need to determine which host is the vSAN Witness Host:
-

```
PS C:\> $WitnessHost = Get-VsanClusterConfiguration -Cluster $Cluster | Select-Object
WitnessHost
```

- Quick PowerShell Tip. This can also be written this way:

```
PS C:\> $WitnessHost = (Get-VsanClusterConfiguration -Cluster $Cluster).WitnessHost
```

Patching the vSAN Witness Host would look like this:

```
# Get the Cluster Object
$Cluster = Get-Cluster -Name "vSAN"
# Get the vSAN Configuration
$ClusterConfiguration = Get-VsanClusterConfiguration -Cluster $Cluster
# Get the vSAN Witness Host
$WitnessHost = $ClusterConfiguration.WitnessHost

# Write that we're patching the vSAN Witness Host
Write-Host "Updating $WitnessHost"

# Check the patch compliance of the witness host
Test-Compliance -Entity $WitnessHost

# Get a list of non-compliant baselines
$NonCompBase = Get-Compliance -Entity $WitnessHost | Where-Object {$_.Status -ne
"Compliant"}

# Notify that we're not compliant with X baseline(s)
Write-Host $WitnessHost "not compliant with baseline: " $NonCompBase.Baseline.Name

# Enumerate each baseline we're not compliant with and patch the host
Foreach ($NonComp in $NonCompBase.BaseLine) {
  # Report which baseline the vSAN Witness host is being patched with
  Write-Host "Patching $WitnessHost with baseline:" $NonComp.Name
      $WitnessHost | Update-Entity -Baseline (Get-Baseline -Name $NonComp.Name) -
Confirm:$False
    }
}
```

A working copy of this script can be found here: https://github.com/jasemccarty/Vsan-SC2N/blob/master/PatchStretchedCluster.ps1

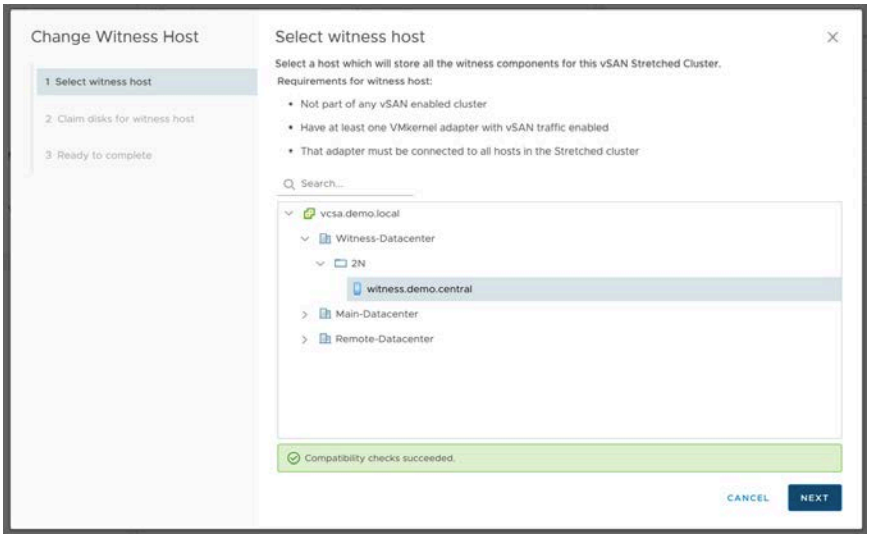
Swapping the vSAN Witness Host

Sometimes a virtualization administrator managing a vSAN Stretched Cluster or 2 Node Cluster will need to swap the vSAN Witness Host.

Scenarios where the vSAN Witness Host might need to be swapped, include configuring a different vSAN Witness Host when the existing vSAN Witness Host is no longer available, or it is desired to use an alternate vSAN Witness Host.

Consider the situation where a vSAN Witness Host has been deleted, corrupted, or can no longer be connected to. For the vSAN Stretched Cluster to be made healthy again, it must have a new vSAN Witness Host join the cluster.

In the vSAN UI, the Change Witness Host Wizard is used to select a new vSAN Witness Host:



Once selected, the disks that will be used for the vSAN Witness Host's disk group are selected, and then the vSAN Witness Host is added to the cluster.

The previous example showed that the Get-VsanClusterConfiguration cmdlet will return the current vSAN Witness Host configuration.

- The Set-VsanClusterConfiguration cmdlet will let us set the new vSAN Witness Host. First let's put the host object into a variable:
-

```
PS C:\> $NewWitness = Get-VMHost -Name "witness2.scdemo.local"
```

- Remember from the New-VsanDiskGroup samples, a disk group has to have a cache and a capacity device at a minimum.
-
- The "Tiny" and "Normal" vSAN Witness Appliances have a 15GB and 350GB vmdk respectively assigned to SCSI(0:1) slot for the use as a capacity device, and they both have a 10GB vmdk assigned to the SCSI(0:2) slot for a cache device.
-

```
PS C:\> $WitnessCapacity = "mpx.vmhba1:C0:T1:L0"
PS C:\> $WitnessCache = "mpx.vmhba1:C0:T2:L0"
```

With vSAN Witness Host parameters, the Set-VsanClusterConfiguration cmdlet can be used to swap the vSAN Witness Host:

```
PS C:\> Set-VsanClusterConfiguration -Configuration $Cluster -WitnessHost $NewWitness -
WitnessHostCacheDisk $WitnessCache -WitnessHostCapacityDisk $WitnessCapacity

Cluster             VsanEnabled  IsStretchedCluster   Last HCL Updated
-------             -----------  ------------------   ----------------
```

```
vSAN          True          True                1/31/19 10:32:00 AM
```

If the previous vSAN Witness Host has been absent for longer than the CLOMD Repair Timer (typically 60 minutes) the vSAN objects will automatically be repaired. In the case where the vSAN Witness Host has been absent less than the CLOMD Repair Timer, the missing vSAN Object Witness Components will be rebuilt automatically when the CLOMD Repair Timer expires.

To force a “Repair Objects Immediately” operation so the vSAN Object Witness Components are immediately recreated, a Get-VsanView call will have to be made. The VsanHealthRepairClusterObjectsImmediately method must be used from the Vsan Health System Managed Object.

```
PS C:\>$VCHS = Get-VsanView -Id "VsanVcClusterHealthSystem-vsan-cluster-health-system"
PS C:\>$VCHS.VsanHealthRepairClusterObjectsImmediate($Cluster.ExtensionData.MoRef,$null)
```

Calling the Repair Objects Immediately method will begin the process to return the vSAN Objects to their Storage Policy Compliant state.

An example of this script can be found here:
https://code.vmware.com/samples/1671

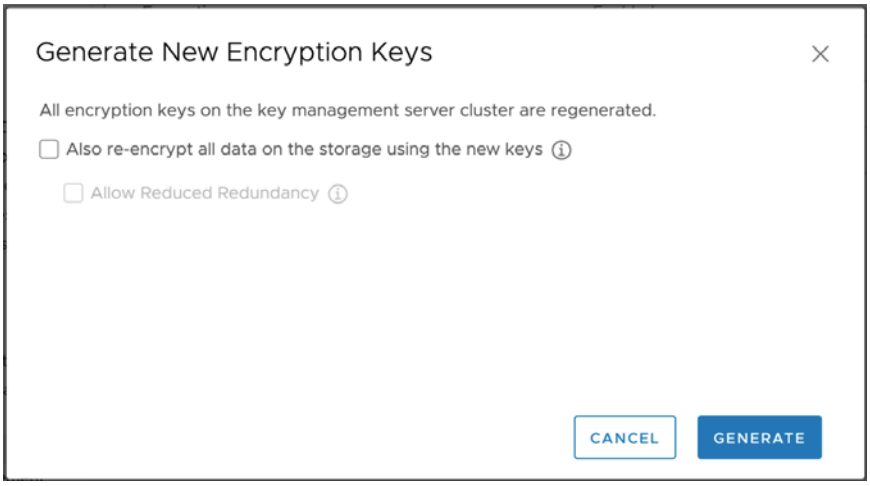
vSAN Encryption Operations

vSAN Encryption tasks that could potentially be performed over time include shallow rekeying, deep rekeying, as well as change the current Key Management Server.

Shallow Rekey

A shallow rekey operation for an encrypted vSAN cluster simply requests a new Key Encryption Key (KEK) from the KMS server, which is then used to re-encrypt the Data Encryption Key (DEK) for each vSAN device.

The vSphere Client has a “Generate New Encryption Keys” option to handle rekey operations.



Pressing “Generate” without selecting the “Also re-encrypt…” checkbox, will perform a shallow rekey.

This operation is not natively available in PowerCLI, so we will have to use Get-VsanView again to perform this task.

```
# Get the Cluster
$Cluster = Get-Cluster -Name “vSANCluster”

# Setup the VsanVcClusterConfigSystem variable
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-system"

# If shallow set to $false, if deep set to $true
$Rekey = $false

# Reduced Redundancy
$Redun = $false

# Perform the Rekey Task
$VsanVcClusterConfig.VsanEncryptedClusterRekey_Task($Cluster.ExtensionData.MoRef,$Rekey,$Redun)
```

There is no requirement for Reduced Redundancy when performing a Shallow Rekey, as no data is moved.

Changing the KMS Server

It could be desirable to change the KMS server that is being used for vSAN Encryption. A typical scenario where an older KMS is being decommissioned and a new one is one such case.

Get-VsanView will be used to set the vSAN Configuration Spec and set the vSAN Cluster to use the new KMS.

```
# Set the VsanVcClusterConfigSystem View
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-
system"

# Setup the KMS Provider Id Specification
$KmsProviderIdSpec = New-Object VMware.Vim.KeyProviderId
```

```
$KmsProviderIdSpec.Id = $KmsClusterProfile.Name

# Setup the Data Encryption Configuration Specification
$DataEncryptionConfigSpec = New-Object VMware.Vsan.Views.VsanDataEncryptionConfig
# Grab the Provider ID for the KMS from vCenter
$DataEncryptionConfig.KmsProviderId = (Get-KmsCluster -Name "NEWKMS").ExtensionData.ClusterId
# Set Encryption to True
$DataEncryptionConfigSpec.EncryptionEnabled = $true

# Set the Reconfigure Specification to use the Data Encryption Configuration Spec
$vsanReconfigSpec = New-Object VMware.Vsan.Views.VimVsanReconfigSpec
$vsanReconfigSpec.DataEncryptionConfig = $DataEncryptionConfigSpec

# Execute the task of changing the KMS Cluster Profile Being Used
$VsanVcClusterConfig.VsanClusterReconfig($VsanCluster.ExtensionData.MoRef,$vsanReconfigSpec)
```

Just as the KMS value was set in configuring vSAN Encryption, this will update the KMS to a different KMS.

Deep Rekey

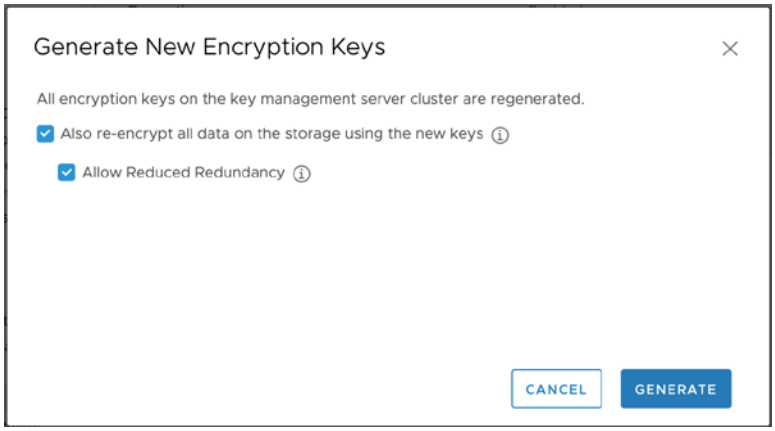
A deep rekey operation for an encrypted vSAN cluster requests a new Key Encryption Key (KEK) from the KMS server, which is then used to re-encrypt the Data Encryption Key (DEK) for each vSAN device.

Once a new KEK is retrieved, the following tasks occur automatically:

- Data is evacuated off of each vSAN Disk Group
- The Disk Group is then removed.
- A new DEK is generated for each device
- The DEK is encrypted with the KEK
- The vSAN Disk Group is recreated
- Data is then moved back to the Disk Group if necessary.

In cases where vSAN has only 2 or 3 nodes, or in cases where there is no way to maintain policy compliance, the Allow Reduced Redundancy option is necessary for the process to complete successfully.

The vSphere Client “Generate New Encryption Keys” option, with the “Also re-encrypt…” checkbox selected, will perform a Deep Rekey. The Allow Reduced Redundancy option becomes available when “Also re-encrypt…” is selected.



This operation is not natively available in PowerCLI, so we will have to use Get-VsanView again to perform this task.

```
# Get the Cluster
$Cluster = Get-Cluster -Name “vSANCluster”

# Setup the VsanVcClusterConfigSystem variable
$VsanVcClusterConfig = Get-VsanView -Id "VsanVcClusterConfigSystem-vsan-cluster-config-system"

# If shallow set to $false, if deep set to $true
$Rekey = $true

# Reduced Redundancy
$Redun = $true

# Perform the Rekey Task
$VsanVcClusterConfig.VsanEncryptedClusterRekey_Task($Cluster.ExtensionData.MoRef,$Rekey,$Redun)
```

Reduced Redundancy may or may not be required when performing a Deep Rekey.

Reporting Recipes

A few sample 'Recipes' are included in this document to detail the process of how one would go about putting together PowerCLI scripts for reporting common aspects of vSAN.

> **Important Note:** The code samples included in this document are not supported by VMware. The code included is only provided as sample code for the purpose of demonstrating different tasks using PowerCLI.

Disk Utilization

A common reporting ask for vSAN, is to be able to report on individual disk, disk group, and individual host capacity utilization.

The Get-VsanDisk cmdlet provides information specific to each disk that is part of a vSAN cluster.

Here is an example of some information returned by Get-VsanDisk:

```
PS /> Get-VsanDisk

CanonicalName        DevicePath                                IsSsd
-------------        ----------                                -----
naa.55cd2e404c166444 /vmfs/devices/disks/naa.55cd2e404c166444 True
naa.55cd2e404c17ba30 /vmfs/devices/disks/naa.55cd2e404c17ba30 True
naa.55cd2e404c17b740 /vmfs/devices/disks/naa.55cd2e404c17b740 True
naa.55cd2e404c17b743 /vmfs/devices/disks/naa.55cd2e404c17b743 True
naa.55cd2e404c166446 /vmfs/devices/disks/naa.55cd2e404c166446 True
naa.55cd2e404c17ba2b /vmfs/devices/disks/naa.55cd2e404c17ba2b True
naa.55cd2e404c17b747 /vmfs/devices/disks/naa.55cd2e404c17b747 True
naa.55cd2e404c17b74b /vmfs/devices/disks/naa.55cd2e404c17b74b True
… continued
```

The disks' names, paths, and type are returned, but there is more. Only displaying the first, additional valuable information is present in the results:

```
PS /> Get-VsanDisk | Select-Object -First 1 | Format-List

ExtensionData           : VMware.Vim.HostScsiDisk
VsanDiskGroup           : Disk group (020000000055cd2e404c166444494e54454c20)
IsCacheDisk             : True
CanonicalName           : naa.55cd2e404c166444
IsSsd                   : True
DevicePath              : /vmfs/devices/disks/naa.55cd2e404c166444
Uuid                    : 020000000055cd2e404c166444494e54454c20
ScsiLun                 : naa.55cd2e404c166444
DiskFormatVersion       : 7
NumComponent            : 0
CapacityGB              : 186.307807445526
UsedPercent             : 0.00000000149965144357321693 00
ReservedPercent         : 0
Name                    : naa.55cd2e404c166444
Id                      : HostSystem-host-115/020000000055cd2e404c166444494e54454c20
Uid                     :
/VIServer=vsphere.local\administrator@vcsa.vcorp.com:443/VsanDisk=HostSystem-host-
115&slash;020000000055cd2e404c166444494e54454
                          c20/
```

CapacityGB and UsedPercent are fields that can be used to show the overall utilization of each disk. Used GB has to be calculated by multiplying the capacity by the percentage used.

```
PS /> $Disk1 = $Cluster | Get-VMHost | Get-VsanDisk | Where-Object {$_.IsCacheDisk -eq
$False} | Select-Object -First 1
PS /> $Disk1.CapacityGB
199.9921875
PS /> $Disk1.UsedPercent
47.857338177272549708973006760
PS /> [math]::abs($Disk1.CapacityGB*($Disk1.UsedPercent/100))
95.710937500000000000000000000
PS />
```

That's just for a single disk though. Each disk could be queried for each disk group, and each disk group for a host, and each host for the cluster.

To do this, loop through the hosts in the cluster, then loop through the disk groups in each host, followed by looping through each disk in the disk group.

```
# Enumerate the cluster and store it
$Cluster = Get-Cluster -Name "vSAN"

# Write the cluster name
Write-Host "Cluster: $Cluster"

# Enumerate the hosts and loop through them
Foreach ($VMHost in ($Cluster|Get-VMHost)) {

  # Write the host we're reporting from
  Write-Host "-$VMHost"

  # Enumerate and Loop through the disk groups
  Foreach ($DiskGroup in ($VMHost | Get-VsanDiskGroup)) {

    # Write the disk group we're reporting from
    Write-Host "--$DiskGroup"

    # Enumerate and store the disks in the current disk group
    $Disks = $DiskGroup | Get-VsanDisk | Where-Object {$_.IsCacheDisk -eq $false}

    # Loop through each of the capacity disks
    Foreach ($Disk in $Disks) {

      # Set the color for the Used Percentage based on our thresholds
      switch ($Disk.UsedPercent) {
        {$_ -ge 0 -and $_ -le 70} {$UsedPctColor="Green"}
        {$_ -ge 70 -and $_ -le 85} {$UsedPctColor="Yellow"}
        {$_ -ge 85 -and $_ -le 101} {$UsedPcColor="Red"}
      }

      # Output the Disk, Capacity, and the Used Percentage
      Write-Host "-- --Disk: $Disk"
      Write-Host "-- -- --Capacity: " $Disk.CapacityGB.ToString("#.##")

      # Calculate used GB by multiplying Capacity by Used %
      $UsedGB = [math]::abs($Disk.CapacityGB*($Disk.UsedPercent/100))
      Write-Host "-- -- --Used GB: " $UsedGB.ToString("#.##")
      Write-Host "-- -- --Used Percent:" $Disk.UsedPercent.ToString("#.##") -
ForegroundColor $UsedPctColor
      Write-Host " "
    }
```

```
  }
}
```

The output looks something like this:

```
PS /> ./vsancapacity.ps1
Cluster: vSAN
-sc1.scdemo.local
--Disk group (02000000006000c291952813a5f8ff33afa05d16d0566972747561)
-- --Disk: naa.6000c2917d44723a60768b58ba3b43f6
-- -- --Capacity:  199.99
-- -- --Used GB:  95.71
-- -- --Used Percent: 47.86

-sc6.scdemo.local
--Disk group (02000000006000c290fa4da20e03ba4b084ca600bb566972747561)
-- --Disk: naa.6000c297483d20353981ce6863548820
-- -- --Capacity:  199.99
-- -- --Used GB:  117.53
-- -- --Used Percent: 58.77

-sc3.scdemo.local
--Disk group (02000000006000c2994a81bb88d7e3aecbe12c5618566972747561)
-- --Disk: naa.6000c29c2b5e7dcba599a777887acdc3
-- -- --Capacity:  199.99
-- -- --Used GB:  126.58
-- -- --Used Percent: 63.29

-sc5.scdemo.local
--Disk group (02000000006000c299bf718e3865d38f4601b62872566972747561)
-- --Disk: naa.6000c2955998f6195970c7d460a48d00
-- -- --Capacity:  199.99
-- -- --Used GB:  133.17
-- -- --Used Percent: 66.59

-sc2.scdemo.local
--Disk group (02000000006000c29ca6b01a0168af2b5c2219fdf7566972747561)
-- --Disk: naa.6000c29acf37e745bbbf8631c9242337
-- -- --Capacity:  199.99
-- -- --Used GB:  119.9
-- -- --Used Percent: 59.95

-sc4.scdemo.local
--Disk group (02000000006000c291ed81e3db71e12f4e2700a825566972747561)
-- --Disk: naa.6000c29e4b94409f19aa16af7295c50c
-- -- --Capacity:  199.99
-- -- --Used GB:  90.62
-- -- --Used Percent: 45.31
PS />
```

Using some additional math, we can serially sum the total capacity and used capacity on a per disk group and per host level:

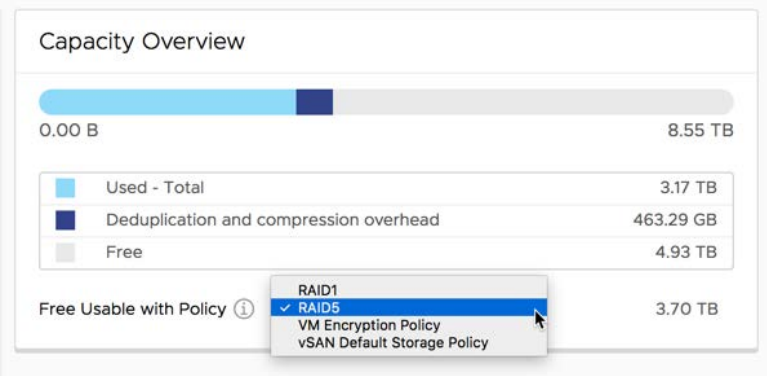
```
# Enumerate the cluster and store it
$Cluster = Get-Cluster -Name “vSAN”

# Write the cluster name
Write-Host "Cluster: $Cluster"

# Enumerate the hosts and loop through them
Foreach ($VMHost in ($Cluster|Get-VMHost)) {

  # Write the host we're reporting from
  Write-Host "-$VMHost"

  # Zero out counter for Disk Groups
  $counter = 0
```

```
  # Zero out Host total
  $HostCapacity = 0
  $HostUsed = 0

  # Enumerate and Loop through the disk groups
  Foreach ($DiskGroup in ($VMHost | Get-VsanDiskGroup)) {

    # Write the disk group we're reporting from
    Write-Host "--($Counter) $DiskGroup"

    # Zero out Disk Group Totals for the current Disk Group
    $DiskGroupCapacity = 0
    $DiskGroupUsed = 0

    # Enumerate and store the disks in the current disk group
    $Disks = $DiskGroup | Get-VsanDisk | Where-Object {$_.IsCacheDisk -eq $false}

    # Loop through each of the capacity disks
    Foreach ($Disk in $Disks) {

      # Set the color for the Used Percentage based on our thresholds
      switch ($Disk.UsedPercent) {
        {$_ -ge 0 -and $_ -le 70} {$UsedPctColor="Green"}
        {$_ -ge 70 -and $_ -le 85} {$UsedPctColor="Yellow"}
        {$_ -ge 85 -and $_ -le 101} {$UsedPcColor="Red"}
      }

      # Output the Disk, Capacity, and the Used Percentage
      Write-Host "-- --Disk: $Disk"

      $DiskGroupCapacity += $Disk.CapacityGB

      Write-Host "-- -- --Capacity: " $Disk.CapacityGB.ToString("#.##")

      # Calculate used GB by multiplying Capacity by Used %
      $UsedGB = [math]::abs($Disk.CapacityGB*($Disk.UsedPercent/100))

      $DiskGroupUsed += $UsedGB

      Write-Host "-- -- --Used GB: " $UsedGB.ToString("#.##")
      Write-Host "-- -- --Used Percent:" $Disk.UsedPercent.ToString("#.##") -
ForegroundColor $UsedPctColor
      Write-Host " "
    }
    Write-Host "Disk Group $Counter Capacity (in
GB):"$DiskGroupCapacity.ToString("#.##")
    Write-Host "Disk Group $Counter Used (in GB):"$DiskGroupUsed.ToString("#.##")

    $HostCapacity += $DiskGroupCapacity
    $HostUsed += $DiskGroupUsed

    $Counter += 1
  }
  Write-Host "-----------------------------------------------------"
  Write-Host "Host Capacity (in GB):"$HostCapacity.ToString("#.##")
  Write-Host "Host Used (in GB):"$HostUsed.ToString("#.##")
  Write-Host ""
  Write-Host "-----------------------------------------------------"
}
```

This script can be found here: https://code.vmware.com/samples/5319

vSAN Capacity Based on Storage Policy
vSAN Datastores have always reported RAW capacity. With the introduction of vSAN 6.7 Update 1, administrators could estimate the amount of useable capacity a vSAN datastore would have available using a given Storage Policy. This is presented in the vSphere Client,

and allows an administrator to choose which policy they wish to estimate using.



Being able to estimate free useable capacity is a great addition to the native reporting of a vSAN cluster. But what if you want to programmatically report this across one or more clusters?

The Get-VsanSpaceUsage cmdlet has provided information including:

- ClusterName
- Cluster Capacity (GB)
- Free Capacity (GB)
- Virtual Disk Usage (GB)
- VM Namespace Disk Usage (GB)
- iSCSI LUN/Target Disk Usage (GB)

Before the release of PowerCLI 11.2, it was not as easy to determine what the usable capacity was when using this Storage Policy, that Storage Policy, or another Storage Policy.

With PowerCLI 11.2, the “VsanWhatIfCapacity” property was added to the Get-VsanSpaceUsage cmdlet.

Invoking the Get-VsanSpaceUsage cmdlet with a -StoragePolicy parameter, will populate the VsanWhatIfCapacity property to easily report what the capacity of the vSAN datastore would be for a given policy:

```
PS /> $DefaultPolicy = Get-SpbmStoragePolicy -Name "vSAN Default Storage Policy"
PS /> $RAID5Policy = Get-SpbmStoragePolicy -Name "RAID5"
PS /> (Get-VsanSpaceUsage -StoragePolicy $DefaultPolicy).VsanWhatIfCapacity
StoragePolicy               TotalWhatIfCapacityGB FreeWhatIfCapacityGB
-------------               --------------------- --------------------
vSAN Default Storage Policy            4471.21875     4201.66796873882

PS /> (Get-VsanSpaceUsage -StoragePolicy $RAID5Policy).VsanWhatIfCapacity
StoragePolicy TotalWhatIfCapacityGB FreeWhatIfCapacityGB
------------- --------------------- --------------------
RAID5               6706.8298017066     6302.50352873374

PS />
```

Per-VM Space Utilization
Per disk, disk group, host, and cluster utilization is a good start, but what about a report that shows the consumption of virtual machines?

The previous script looked at capacity used, on vSAN capacity disks in a vSAN disk group, regardless of the amount, or type of data. Individual VM's and their objects (VM Namespace, Virtual Disks, Swap Files, etc) can be reported on as well, though not natively with a PowerCLI cmdlet.

William Lam put together a sample script to report detailed per-VM space utilization that utilizes VsanQueryObjectIdentities API method. This is a method that is available as part of the VsanObjectSystem Managed Object.

The VsanObjectSystem Managed Object can be used with vCenter or individual vSAN hosts.

In William's script, the VsanObjectSystem is used against individual hosts to return raw data from the VsanQueryObjectIdentities method.

The documentation suggests that using the VsanQueryObjectIdentities method against the cluster will return the same information as using the method against individual hosts. As of this writing, PowerCLI 11.1 does not return full complement of data from this method.

When calling the VsanQueryObjectIdentities method against a single host, only the objects that part of VM's that are are registered on that host are retrieved.

Because of this, each host will be queried for VM's and their objects to return the per-VM utilization. A quick script will retrieve the data that will be used to report on.

```
# Scope query within vSAN/vSphere Cluster
$clusterView = Get-View -ViewType ClusterComputeResource -Property Name,Host -Filter
@{"name"=$Cluster}

# Retrieve list of ESXi hosts from cluster
# which we will need to directly connect to use call VsanQueryObjectIdentities()
$vmhosts = $clusterView.host

# Setup our results array which results from each host will be aggregated
$results = @()

foreach ($vmhost in $vmhosts) {
  # Get the VMHost Name
  $vmhostView = Get-View $vmhost -Property Name

  # Connect to the VMHost
  $esxiConnection = Connect-VIServer -Server $vmhostView.name -User $ESXiHostUsername -
Password $ESXiHostPassword

  # Connect to the VsanObjectSystem Managed Object on the current host
  $vos = Get-VSANView -Id "VsanObjectSystem-vsan-object-system" -Server $esxiConnection

  # Retrieve the vSAN Object identities of the objects on the VMHost
  $identities = $vos.VsanQueryObjectIdentities($null,$null,$null,$false,$true,$true)

  # convert the raw Json formatted data to something more useful
```

```
  $json = $identities.RawData|ConvertFrom-Json
  $jsonResults = $json.identities.vmIdentities
}
```

The converted identity data from each host looks something like this:

```
@{vmNsObjectUuid=4ba41a5c-9c12-0643-9921-ecf4bbf0d200; vmInstanceUuid=42027559-0f29-
03cb-260e-c737f79dd20f; objIdentities=System.Object[]} @{vmNsObjectUuid=8ba41a5c-2564-
bbb5-3feb-ecf4bbf0ba08; vmInstanceUuid=42023b87-c24b-7660-88d4-1eab8b9e98c6;
objIdentities=System.Object[]} @{vmNsObjectUuid=82a41a5c-943c-a8d9-cff7-ecf4bbf0ba08;
vmInstanceUuid=420239a9-69f0-635b-3f60-5e3957a8a1ba; objIdentities=System.Object[]}
@{vmNsObjectUuid=b5a41a5c-e296-7399-4973-ecf4bbf0d200; vmInstanceUuid=4202bf58-28b5-
5abd-456a-2264d5feeddc; objIdentities=System.Object[]} @{vmNsObjectUuid=80a41a5c-8426-
33bd-9c9f-ecf4bbf0d200; vmInstanceUuid=420203b1-5e39-bf2e-dc53-45ff41893243;
objIdentities=System.Object[]} @{vmNsObjectUuid=a1a41a5c-767c-911b-a793-ecf4bbf0b8d8;
vmInstanceUuid=42020e78-4b16-8c8d-6b43-c59836f89455; objIdentities=System.Object[]}
@{vmNsObjectUuid=70a41a5c-149e-be12-123f-ecf4bbf0d200; vmInstanceUuid=4202337b-f91d-
8eb4-b789-2d7194ce0872; objIdentities=System.Object[]} @{vmNsObjectUuid=d7bafe5b-f1b9-
fcf9-35aa-ecf4bbf0ba08; vmInstanceUuid=4202626c-d2c3-4e9d-ca56-f1bb4ec8808c;
objIdentities=System.Object[]}
```

The $jsonResults are not really that easy to read.

A loop can be used to enumerate all of the identities, retaining those that provide some value, and discarding those that do not. The retained results can then be massaged to have a more friendly output.

```
foreach ($vmInstance in $jsonResults) {
  # Here we're only grabbing the objIdentities in the $jsonResults
  $identities = $vmInstance.objIdentities

  # Loop through each of the Identities, sorting on the Property type
  foreach ($identity in $identities | Sort-Object -Property "type") {

    # Retrieve the VM Name
    if($identity.type -eq "namespace") {

      # Remember that we're not connected to vCenter anymore, only the current host
      # We'll need to retrive the attributes of the object from the current host
      $vsanIntSys = Get-View (Get-VMHost -Server
$esxiConnection).ExtensionData.ConfigManager.vsanInternalSystem

      # Setup the $attributes variable so we can enumerate all of the object attributes
      $attributes = ($vsanIntSys.GetVsanObjExtAttrs($identity.uuid)) | ConvertFrom-JSON

      # Loop through each of the object's attributes, and retrieve the VM's name
      foreach ($attribute in $attributes | Get-Member) {
        # crappy way to iterate through keys ...
        if($($attribute.Name) -ne "Equals" -and $($attribute.Name) -ne "GetHashCode" -
and $($attribute.Name) -ne "GetType" -and $($attribute.Name) -ne "ToString") {
          $objectID = $attribute.name
          $vmName = $attributes.$($objectID).'User friendly name'
        }
      }
    }

    # Other attributes returned include physicalUsedCapacity, reservedCapacity
    # File path, and the  type of object

    # Convert B to GB
    $physicalUsedGB = [math]::round($identity.physicalUsedB/1GB, 2)
    $reservedCapacityGB = [math]::round($identity.reservedCapacityB/1GB, 2)

    # Build our custom object to store only the data we care about
    $tmp = [pscustomobject] @{
```

```
      VM = $vmName
      File = $identity.description;
      Type = $identity.type;
      physicalUsedGB = $physicalUsedGB;
      reservedCapacityGB = $reservedCapacityGB;
    }

    # Filter out a specific VM if provided
      if($VM) {
        if($vmName -eq $VM) {
          $results += $tmp
        }
      } else {
        $results += $tmp
      }
    }
  }
```

Combining the retrieval of identity data from each host, along with the cleaned-up reporting gives us an output something like this:

```
VM    File                                                                Type       physUsedGB rsrvdGB
--    ----                                                                ----       ---------- -------
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST.vmx       namespace  0.71       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_11.vmdk   vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_2.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_4.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_3.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_10.vmdk   vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_7.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_6.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_8.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_5.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_1.vmdk    vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_13.vmdk   vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST.vmdk      vdisk      1.14       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_15.vmdk   vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_12.vmdk   vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_9.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_14.vmdk   vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST.vswp      vmswap     0.04       0
DC2   [vsanDatastore] 23b1b65b-e473-60ba-c119-ecf4bbf0ba08/DC2.vmx        namespace  0.81       0
DC2   [vsanDatastore] 23b1b65b-e473-60ba-c119-ecf4bbf0ba08/DC2.vmdk       vdisk     56.47       0
DC2   [vsanDatastore] 23b1b65b-e473-60ba-c119-ecf4bbf0ba08/DC2.vswp       vmswap     0.04       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2.vmx        namespace  0.82       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2_1.vmdk     vdisk     32.36       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2_2.vmdk     vdisk      0.07       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2.vmdk       vdisk      1.25       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2.vswp       vmswap     0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST.vmx       namespace  0.71       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_11.vmdk   vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_2.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_4.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_3.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_10.vmdk   vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_7.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_6.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_8.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_5.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_1.vmdk    vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_13.vmdk   vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST.vmdk      vdisk      1.14       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_15.vmdk   vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_12.vmdk   vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_9.vmdk    vdisk      0.04       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST_14.vmdk   vdisk      0.07       0
TEST  [vsanDatastore] 014b3e5c-11ad-ffa3-bbc3-ecf4bbf0d3c8/TEST.vswp      vmswap     0.04       0
DC2   [vsanDatastore] 23b1b65b-e473-60ba-c119-ecf4bbf0ba08/DC2.vmx        namespace  0.81       0
DC2   [vsanDatastore] 23b1b65b-e473-60ba-c119-ecf4bbf0ba08/DC2.vmdk       vdisk     56.47       0
DC2   [vsanDatastore] 23b1b65b-e473-60ba-c119-ecf4bbf0ba08/DC2.vswp       vmswap     0.04       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2.vmx        namespace  0.82       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2_1.vmdk     vdisk     32.36       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2_2.vmdk     vdisk      0.07       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2.vmdk       vdisk      1.25       0
SC2_1 [vsanDatastore] 75a41a5c-3d77-56cb-6e26-ecf4bbf0d3c8/SC2.vswp       vmswap     0.04       0
```

The full source of the combined code can be found here:

https://github.com/lamw/vghetto-scripts/blob/master/powershell/VSANVMDetailedUsage.ps1

Per-VM Storage Policy Compliance
Failure to maintain Storage Policy compliance for vSAN Objects could determine the difference from available data and inaccessible data.

vSAN Objects that are compliant with their Storage Policy are happy and healthy vSAN Objects. PowerCLI has long had the ability to report the Storage Policy compliance status of vSAN Objects using the Get-SpbmEntityConfiguration cmdlet.

```
PS /Users/jase/PowerCLI> Get-SpbmEntityConfiguration

Entity                         Storage Policy                 Status          Time Of Check
------                         --------------                 ------          -------------
VCSA                           RAID5                          compliant       1/19/19 4:06:21 AM
SQL1                           RAID5                          compliant       1/19/19 4:06:21 AM
VCSA1                          RAID5                          compliant       1/19/19 4:06:21 AM
APP7                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:22 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:22 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:22 AM
APP6                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:23 AM
Hard disk 2                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:23 AM
Hard disk 2                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:23 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:22 AM
Hard disk 2                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:24 AM
Hard disk 2                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:24 AM
APP1                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:24 AM
APP5                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:24 AM
NEWVM                          Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:24 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:25 AM
APP3                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:25 AM
APP4                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:25 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:25 AM
APP2                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:25 AM
Hard disk 3                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:26 AM
Hard disk 3                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:26 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:26 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:26 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:26 AM
TEST                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:26 AM
Hard disk 1                    Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:06:27 AM
Hard disk 2                    vSAN Default Storage Policy    compliant       1/19/19 4:06:27 AM
Hard disk 1                    vSAN Default Storage Policy    compliant       1/19/19 4:06:27 AM
Hard disk 1                    vSAN Default Storage Policy    compliant       1/19/19 4:06:27 AM
Hard disk 3                    vSAN Default Storage Policy    compliant       1/19/19 4:06:28 AM
Hard disk 1                    vSAN Default Storage Policy    compliant       1/19/19 4:06:28 AM
```

Using the cmdlet by itself will return the Storage Policy information for every object that is managed by a vCenter Server. Also, there is no easy way to tell from the above information which “Hard disk 1” belongs to which VM.

It should be as easy as only retrieving the VMs from the $Cluster right?

```
PS /> $Cluster | Get-VM | Sort-Object -Property Name | Get-SpbmEntityConfiguration

Entity                         Storage Policy                 Status          Time Of Check
------                         --------------                 ------          -------------
APP1                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:02 AM
APP2                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:02 AM
APP3                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:03 AM
APP4                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:04 AM
APP5                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:05 AM
APP6                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:06 AM
APP7                           Mirrored-75-SpaceReserved-5... compliant       1/19/19 4:20:06 AM
SQL1                           RAID5                          compliant       1/19/19 4:20:07 AM
VCSA                           RAID5                          compliant       1/19/19 4:20:08 AM
VCSA1                          RAID5                          compliant       1/19/19 4:20:09 AM
```

Wrong. In this example, only the VM's are the Entities, not their disks. Complete results will require a bit more elaborate code.

The above code already returns the compliance status of the VM's namespace. Retrieving the hard disks from each VM is relatively easy but will require placing all of the VMs in an array, and then looping through that array to query the VM for a list of hard disks that can then be checked for compliance.

```
$VMS = $Cluster | Get-VM | Sort-Object Name
Foreach ($VM in $VMS) {
    Get-SpbmEntityConfiguration -VM $VM
    $HardDisks = Get-HardDisk -VM $VM
    Foreach ($HardDisk in $HardDisks) {
        Get-SpbmEntityConfiguration -HardDisk $HardDisk
    }
}
```

This results in:

```
Entity                    Storage Policy                Status          Time Of Check
------                    --------------                ------          -------------
APP1                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:36 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:37 AM
Hard disk 2               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:38 AM
APP2                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:39 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:40 AM
APP3                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:41 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:42 AM
Hard disk 2               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:43 AM
Hard disk 3               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:44 AM
APP4                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:44 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:45 AM
APP5                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:46 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:48 AM
Hard disk 2               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:49 AM
APP6                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:49 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:50 AM
APP7                      Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:51 AM
Hard disk 1               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:52 AM
Hard disk 2               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:53 AM
Hard disk 3               Mirrored-75-SpaceReserved-5... compliant      1/19/19 4:50:54 AM
SQL1                      RAID5                         compliant       1/19/19 4:50:55 AM
Hard disk 1               vSAN Default Storage Policy   compliant       1/19/19 4:50:56 AM
VCSA                      RAID5                         compliant       1/19/19 4:50:57 AM
Hard disk 1               vSAN Default Storage Policy   compliant       1/19/19 4:50:58 AM
Hard disk 2               vSAN Default Storage Policy   compliant       1/19/19 4:50:59 AM
Hard disk 3               vSAN Default Storage Policy   compliant       1/19/19 4:51:00 AM
VCSA1                     RAID5                         compliant       1/19/19 4:51:01 AM
Hard disk 1               vSAN Default Storage Policy   compliant       1/19/19 4:51:02 AM
```

What if this information is exported as a CSV file that is imported into Excel or some other location? "Hard disk 1" is listed multiple times. Using a $results array like in William's example above, with an object for the VM Namespace results and an object for the Hard Disk results, the Entity results can be modified to be a bit more descriptive:

```
$VMs = $Cluster | Get-VM | Sort-Object Name

$results = @()
Foreach ($VM in $VMs) {
    $VmCompliance = Get-SpbmEntityConfiguration -VM $VM
    $VmTmp = [PSCustomObject] @{
        VM = $VmCompliance.Name + "-Namespace     "
        Policy = $VmCompliance.StoragePolicy
        Status = $VmCompliance.ComplianceStatus
        TimeofCheck = $VmCompliance.TimeOfCheck
    }

    $results += $VmTmp
```

```
    $HardDisks = Get-HardDisk -VM $VM
    Foreach ($HardDisk in $HardDisks) {
        $HdCompliance = Get-SpbmEntityConfiguration -HardDisk $HardDisk

        $HdTmp = [PSCustomObject] @{
            VM = $VmCompliance.Name + "-" + $HdCompliance.Name
            Policy = $HdCompliance.StoragePolicy
            Status = $HdCompliance.ComplianceStatus
            TimeofCheck = $HdCompliance.TimeOfCheck
        }
        $results +=$HdTmp
    }
}
$results |ft
```

Which results in:

```
VM                 Policy                              Status TimeofCheck
--                 ------                              ------ -----------
APP1-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:05 AM
APP1-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:06 AM
APP1-Hard disk 2   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:06 AM
APP2-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:07 AM
APP2-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:07 AM
APP3-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:08 AM
APP3-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:09 AM
APP3-Hard disk 2   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:09 AM
APP3-Hard disk 3   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:10 AM
APP4-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:10 AM
APP4-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:11 AM
APP5-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:11 AM
APP5-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:12 AM
APP5-Hard disk 2   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:12 AM
APP6-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:13 AM
APP6-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:13 AM
APP7-Namespace     Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:14 AM
APP7-Hard disk 1   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:15 AM
APP7-Hard disk 2   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:15 AM
APP7-Hard disk 3   Mirrored-75-SpaceReserved-5K-Limit compliant 1/19/19 4:55:16 AM
SQL1-Namespace     RAID5                               compliant 1/19/19 4:55:16 AM
SQL1-Hard disk 1   vSAN Default Storage Policy         compliant 1/19/19 4:55:17 AM
VCSA-Namespace     RAID5                               compliant 1/19/19 4:55:17 AM
VCSA-Hard disk 1   vSAN Default Storage Policy         compliant 1/19/19 4:55:18 AM
VCSA-Hard disk 2   vSAN Default Storage Policy         compliant 1/19/19 4:55:19 AM
VCSA-Hard disk 3   vSAN Default Storage Policy         compliant 1/19/19 4:55:19 AM
VCSA1-Namespace ... RAID5                              compliant 1/19/19 4:55:19 AM
VCSA1-Hard disk 1  vSAN Default Storage Policy         compliant 1/19/19 4:55:20 AM
```

Sample RVC vsan.vm_object_info Report

The Ruby vSphere Console (RVC) has been a tool that vSAN administrators used early on for administration and reporting purposes.

The RVC requires administrators to connect to vCenter over SSH, connect to the RVC for the current instance, and call different scripts to retrieve different results.

These RVC scripts are very valuable, but do not easily allow any interaction with other scripting processes.

With the introduction of the Get-VsanObject and Get-VsanComponent cmdlets in PowerCLI 11.0, much of the information returned from RVC scripts can be returned with PowerCLI.

```
# Requires William Lam's VSANUUIDtoVM Function
# https://github.com/lamw/vghetto-scripts/blob/master/powershell/VSANUUIDTranslate.ps1

# Get the vSAN Cluster Object
$Cluster = Get-Cluster -Name "vSAN"
```

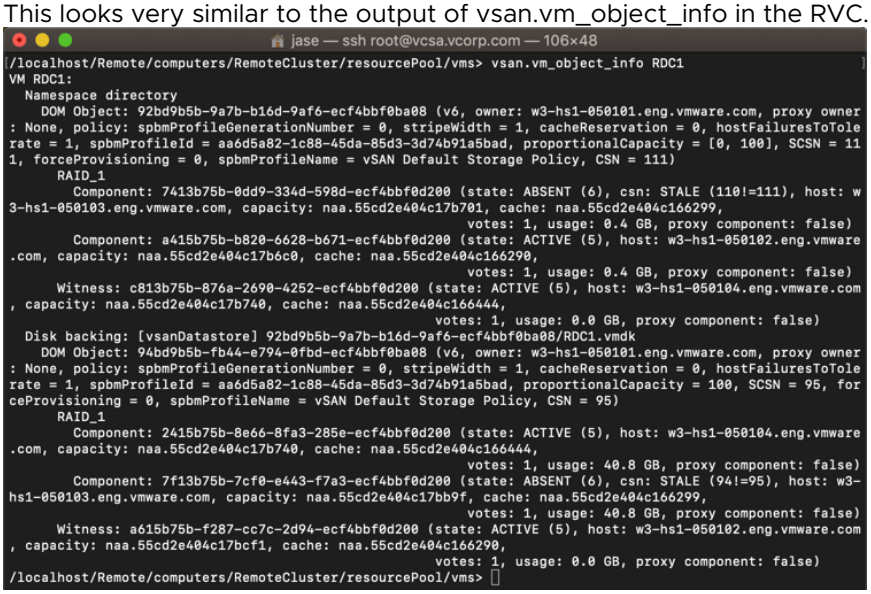
```
# Get the working VM to report on
$VsanVM = Get-VM -Name $VM

# Get the vSAN Objects associated with the VM
$VMObjects = Get-VsanObject -VM $VsanVM

#
    $SPBM_Policies = Get-SpbmStoragePolicy
    Write-Host "VM $VsanVM"

Foreach ($VMObject in $VMObjects) {

  Switch ($VMObject.Type) {
    "VmNamespace" { Write-Host "Namespace directory - " -NoNewline}
    "VmSwap" { Write-Host "VmSwap - " -NoNewline }
    "VDisk" { Write-Host "Virtual Disk - " -NoNewline }
}

Write-Host "Storage Policy - " $VMObject.StoragePolicy.Name -NoNewline
Write-Host (Get-SpbmStoragePolicy -Name
$VMObject.StoragePolicy).AnyOfRuleSets.AnyOfRuleSets
        Get-VSANUUIDToVM -Cluster $Cluster -VSANObjectID $VMObject.id

  Foreach ($VsCp in (Get-VsanComponent -VsanObject $VMObject)) {

    Write-Host "   " $VsCP.Type ": " -NoNewline
    Write-Host $VsCp.id " (" -NoNewline
    Write-Host "state:" -NoNewline
    If ($VsCp.Status -eq "ACTIVE") {
      Write-Host $VsCp.Status ", " -NoNewline -ForegroundColor Green
    } else {
      Write-Host $VsCp.Status ", " -NoNewline -ForegroundColor Yellow
    }
    Write-Host "capacity disk:" $VsCp.VsanDisk -NoNewline
    Write-Host " host: " $VsCp.VsanDisk.VsanDiskGroup.VMHost.Name  -NoNewline
    Write-Host ")"
  }
  Write-Host " "
  }
```

The output looks very much like the RVC vsan.vm_object_info script:

```
PS /> Get-SomeVsanObjectInformation -VsanCluster "vSAN" -VM APP2
VM APP2
Namespace directory - Storage Policy -  Mirrored-75-SpaceReserved-5K-Limit

UUID                : b8224a5c-c6c6-5ce9-7eeb-005056826867
Object type         : vsan
Object size         : 273804165120
User friendly name  : APP2_1
HA metadata         : (null)
Allocation type     : Thin
Policy              : (("stripeWidth" i1) ("cacheReservation" i0)
("proportionalCapacity" i75) ("hostFailuresToTolerate" i1) ("forceProvisioning" i0)
                      ("spbmProfileId" "5302e23b-b12f-4e0a-a3ef-51c74fcfedba")
("spbmProfileGenerationNumber" l+1) ("replicaPreference" "Performance")
                      ("iopsLimit" i5000) ("checksumDisabled" i0) ("spbmProfileName"
"Mirrored-75-SpaceReserved-5K-Limit"))
Object class        : vmnamespace
Object capabilities : NONE
Object path         : /vmfs/volumes/vsan:52c4a44798101504-e2bdb0a7f4ca8b50/APP2_1
Group uuid          : 00000000-0000-0000-0000-000000000000
Container uuid      : 00000000-0000-0000-0000-000000000000

    Component : 7f4d4b5c-15df-b4e5-d55d-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c2917d44723a60768b58ba3b43f6 host:  sc1.scdemo.local)
    Component : f04d4b5c-fc36-3488-1a13-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c297483d20353981ce6863548820 host:  sc6.scdemo.local)
```

```
    Witness : f04d4b5c-930f-3888-0630-005056822e0f  (state:A ACTIVE, capacity disk:
naa.6000c29acf37e745bbbf8631c9242337 host:  sc2.scdemo.local)

Virtual Disk - Storage Policy -  Mirrored-75-SpaceReserved-5K-Limit
UUID                : bb224a5c-8536-5d86-9f3d-005056826867
Object type         : vsan
Object size         : 42949672960
User friendly name  : (null)
HA metadata         : (null)
Allocation type     : Thin
Policy              : (("stripeWidth" i1) ("cacheReservation" i0)
("proportionalCapacity" i75) ("hostFailuresToTolerate" i1) ("forceProvisioning" i0)
                      ("spbmProfileId" "5302e23b-b12f-4e0a-a3ef-51c74fcfedba")
("spbmProfileGenerationNumber" l+1) ("replicaPreference" "Performance")
                      ("iopsLimit" i5000) ("checksumDisabled" i0) ("spbmProfileName"
"Mirrored-75-SpaceReserved-5K-Limit"))
Object class        : vdisk
Object capabilities : NONE
Object path         : /vmfs/volumes/vsan:52c4a44798101504-e2bdb0a7f4ca8b50/b8224a5c-
c6c6-5ce9-7eeb-005056826867/APP2.vmdk
Group uuid          : b8224a5c-c6c6-5ce9-7eeb-005056826867
Container uuid      : (null)

    Component : cf484f5c-3fcc-3fc7-c88b-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c29acf37e745bbbf8631c9242337 host:  sc2.scdemo.local)
    Component : cf484f5c-1c32-42c7-e024-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c2917d44723a60768b58ba3b43f6 host:  sc1.scdemo.local)
    Witness : cf484f5c-0057-43c7-cf25-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c297483d20353981ce6863548820 host:  sc6.scdemo.local)

VmSwap - Storage Policy -  Mirrored-75-SpaceReserved-5K-Limit
UUID                : a6744a5c-4653-33e2-b892-00505682bbb0
Object type         : vsan
Object size         : 4294967296
User friendly name  : (null)
HA metadata         : (null)
Allocation type     : Thin
Policy              : (("stripeWidth" i1) ("cacheReservation" i0)
("proportionalCapacity" i75) ("hostFailuresToTolerate" i1) ("forceProvisioning" i1)
                      ("spbmProfileId" "5302e23b-b12f-4e0a-a3ef-51c74fcfedba")
("spbmProfileGenerationNumber" l+1) ("replicaPreference" "Performance")
                      ("iopsLimit" i5000) ("checksumDisabled" i0) ("spbmProfileName"
"Mirrored-75-SpaceReserved-5K-Limit"))
Object class        : vmswap
Object capabilities : NONE
Object path         : /vmfs/volumes/vsan:52c4a44798101504-e2bdb0a7f4ca8b50/b8224a5c-
c6c6-5ce9-7eeb-005056826867/APP2-d5c0da44.vswp
Group uuid          : b8224a5c-c6c6-5ce9-7eeb-005056826867
Container uuid      : (null)

    Component : cf484f5c-744d-72b1-798d-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c297483d20353981ce6863548820 host:  sc6.scdemo.local)
    Component : cf484f5c-247d-75b1-6d2d-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c29acf37e745bbbf8631c9242337 host:  sc2.scdemo.local)
    Witness : cf484f5c-9bb9-76b1-ca52-005056822e0f  (state:ACTIVE, capacity disk:
naa.6000c2917d44723a60768b58ba3b43f6 host:  sc1.scdemo.local)
```

This looks very similar to the output of vsan.vm_object_info in the RVC.



This script can be found here: https://code.vmware.com/samples/4710

vSAN Encryption Health
An Encryption Health Report can be used to quickly see the state of an encrypted vSAN cluster.

This information is not directly accessible using a PowerCLI cmdlet. The Get-VsanView cmdlet will be used retrieve the encryption health of the cluster.

```
# Enumerate the cluster and store it
$Cluster = Get-Cluster -Name “vSAN”

# vSAN Cluster Health
$VsanHealth = Get-VsanView -Id VsanVcClusterHealthSystem-vsan-cluster-health-system

#Grab our Encryption Health Information
$EncryptionHealth = $VsanHealth.VsanQueryVcClusterHealthSummary(
$Cluster.ExtensionData.MoRef,$null,$null,$null,
@('encryptionHealth'),$null,"defaultView").EncryptionHealth
```

The $EncryptionHealth variable includes the following information:

```
PS /> $EncryptionHealth

OverallHealth : green
ConfigHealth  : green
KmsHealth     : green
VcKmsResult   : VMware.Vsan.Views.VsanVcKmipServersHealth
HostResults   : {sc5.scdemo.local, sc4.scdemo.local, sc1.scdemo.local,
sc3.scdemo.local...}
AesniHealth   : green
```

Immediately some items are values available that can be reported on,

including the overall health, the configuration health, and the AES-NI Health. Others must be expanded upon.

The VcKmsResult property can be expanded to dive deeper into the health of the KMS Configuration:

```
PS /> $EncryptionHealth.VcKmsResult

Health               : green
Error                :
KmsProviderId        : KMS
KmsHealth            : {10.127.75.113}
ClientCertHealth     : green
ClientCertExpireDate : 11/7/22 9:41:06 AM
```

More results can be found within the HostResults property for each host in the cluster.

```
PS /> $EncryptionHealth.HostResults

Hostname         : sc5.scdemo.local
EncryptionInfo   : VMware.Vsan.Views.VsanHostEncryptionInfo
OverallKmsHealth : green
KmsHealth        : {10.127.75.113}
AesniEnabled     : True

Hostname         : sc4.scdemo.local
EncryptionInfo   : VMware.Vsan.Views.VsanHostEncryptionInfo
OverallKmsHealth : green
KmsHealth        : {10.127.75.113}
DiskResults      : {VMware.Vsan.Views.VsanDiskEncryptionHealth,
VMware.Vsan.Views.VsanDiskEncryptionHealth}
AesniEnabled     : True

Hostname         : sc1.scdemo.local
EncryptionInfo   : VMware.Vsan.Views.VsanHostEncryptionInfo
OverallKmsHealth : green
KmsHealth        : {10.127.75.113}
DiskResults      : {VMware.Vsan.Views.VsanDiskEncryptionHealth,
VMware.Vsan.Views.VsanDiskEncryptionHealth}
AesniEnabled     : True

Hostname         : sc3.scdemo.local
EncryptionInfo   : VMware.Vsan.Views.VsanHostEncryptionInfo
OverallKmsHealth : green
KmsHealth        : {10.127.75.113}
DiskResults      : {VMware.Vsan.Views.VsanDiskEncryptionHealth,
VMware.Vsan.Views.VsanDiskEncryptionHealth}
AesniEnabled     : True

Hostname         : sc2.scdemo.local
EncryptionInfo   : VMware.Vsan.Views.VsanHostEncryptionInfo
OverallKmsHealth : green
KmsHealth        : {10.127.75.113}
DiskResults      : {VMware.Vsan.Views.VsanDiskEncryptionHealth,
VMware.Vsan.Views.VsanDiskEncryptionHealth}
AesniEnabled     : True

Hostname         : sc6.scdemo.local
EncryptionInfo   : VMware.Vsan.Views.VsanHostEncryptionInfo
OverallKmsHealth : green
KmsHealth        : {10.127.75.113}
DiskResults      : {VMware.Vsan.Views.VsanDiskEncryptionHealth,
VMware.Vsan.Views.VsanDiskEncryptionHealth}
AesniEnabled     : True
```

The DiskResults sub-property shows Disk Health information. Looking closely at one of the hosts

```
PS /> $Host4 = ($EncryptionHealth.HostResults | Where-Object {$_.DiskResults} | Where-
Object {$_.Hostname -eq "sc4.scdemo.local"})
PS /> $Host4.DiskResults

DiskHealth                                EncryptionIssues
----------                                ----------------
VMware.Vsan.Views.VsanPhysicalDiskHealth
VMware.Vsan.Views.VsanPhysicalDiskHealth
```

Diving further into the disk health entries (abbreviated for space) shows disk encryption related information:

```
PS /> $Host4.DiskResults.DiskHealth

Name                        : naa.6000c29e4b94409f19aa16af7295c50c
Uuid                        : 5263794a-f740-8718-0e91-b0ba6bf5a56a
EncryptionEnabled           : True
KmsProviderId               : KMS
KekId                       : 3e78ba61-20c9-11e9-937e-005056011b78
DekGenerationId             : 9
EncryptedUnlocked           : True

Name                        : naa.6000c291ed81e3db71e12f4e2700a825
Uuid                        : 52a3cff5-8ad3-a59e-0f2f-ec9f13874329
EncryptionEnabled           : True
KmsProviderId               : KMS
KekId                       : 3e78ba61-20c9-11e9-937e-005056011b78
DekGenerationId             : 9
EncryptedUnlocked           : True
```

Also, encryption information for the host can be retrieved.

```
PS /> $Host4.EncryptionInfo

Enabled             : True
KekId               : 3e78ba61-20c9-11e9-937e-005056011b78
HostKeyId           : c5060727-19dd-11e9-937e-005056011b78
KmipServers         : {VMware.Vim.KeyProviderId}
KmsServerCerts      : {CC:20:75:D8:3A:D1:2C:94:D9:CD:75:3F:5D:D9:02:B0:CF:BD:BB:9F}
ClientKey           : 2dac1e2746b1726f56cb3d72c0f935aee158c4aa
ClientCert          : 7B:FD:FD:AA:DF:C2:98:C6:3B:C7:D4:47:1A:35:4E:EE:A2:95:A5:2D
DekGenerationId     : 9
Changing            : False
EraseDisksBeforeUse : False
```

Each of these can be easily reported on.

Creating a report to show the encryption state of a vSAN cluster can pull each of these pieces of information together.

```
PS /> $Host4.EncryptionInfo

Enabled             : True
KekId               : 3e78ba61-20c9-11e9-937e-005056011b78
HostKeyId           : c5060727-19dd-11e9-937e-005056011b78
KmipServers         : {VMware.Vim.KeyProviderId}
KmsServerCerts      : {CC:20:75:D8:3A:D1:2C:94:D9:CD:75:3F:5D:D9:02:B0:CF:BD:BB:9F}
ClientKey           : 2dac1e2746b1726f56cb3d72c0f935aee158c4aa
```

```
ClientCert          : 7B:FD:FD:AA:DF:C2:98:C6:3B:C7:D4:47:1A:35:4E:EE:A2:95:A5:2D
DekGenerationId     : 9
Changing            : False
EraseDisksBeforeUse : False
```

```
# Enumerate the cluster and store it
$Cluster = Get-Cluster -Name “vSAN”

# vSAN Cluster Health
$VsanHealth = Get-VsanView -Id VsanVcClusterHealthSystem-vsan-cluster-health-system

#Grab our Encryption Health Information
$EncryptionHealth = $VsanHealth.VsanQueryVcClusterHealthSummary(
$Cluster.ExtensionData.MoRef,$null,$null,$null,
@('encryptionHealth'),$null,"defaultView").EncryptionHealth

# Output the General Health of the Cluster
Write-Host " Overall Health:        "$EncryptionHealth.OverallHealth
Write-Host " Configuration Health:  "$EncryptionHealth.ConfigHealth
Write-Host " KMS Health:            "$EncryptionHealth.KmsHealth
Write-Host " KMS Server:            "$EncryptionHealth.VcKmsResult.KmsProviderId
Write-Host "------------------------------------------------------------------------"
Write-Host " Per Host Results"
Write-Host "------------------------------------------------------------------------"
ForEach ($VMHost in ($Cluster|Get-VMHost|Sort-Object -Property Name)) {
  $HostHealth = $EncryptionHealth.HostResults | Where-Object {$_.Hostname -eq $VMHost}
  Write-Host "****** Host:"$VMHost
  Write-Host "       Overall KMS Health:    "$HostHealth.OverallKmsHealth
  Write-Host "       AES-NI Enabled:        "$HostHealth.AesniEnabled
  Write-Host "       Disk Status:"

  Foreach ($Disk in $HostHealth.DiskResults.DiskHealth) {
    Write-Host "       * Disk:                "$Disk.Name
    Write-Host "       ** Encryption Enabled: "$Disk.EncryptionEnabled
    Write-Host "       ** KMS:                "$Disk.KmsProviderId
    Write-Host "       ** KekId:              "$Disk.KekId
    Write-Host "       ** DekGeneration:      "$Disk.DekGenerationId
  }
  Write-Host " ------------------------------------------------------------------------"
}
```

A more elaborate version of this script can be found at https://code.vmware.com/samples/2783

Document Summary

The code examples listed in this document are for the purpose of illustrating some capabilities using PowerCLI with the vSAN. These code examples are not supported by VMware.

Be sure to visit the VMware Code at https://code.vmware.com for more examples of using PowerCLI with vSphere and vSAN.

References

Additional Documentation

For more information about VMware vSAN, please visit the product pages at http://www.vmware.com/products/vsan.html

Below are some links to online documentation:

Storage Hub:
https://storagehub.vmware.com/

Virtual Blocks:
http://virtualblocks.com/

VMware vSAN Community:
https://communities.vmware.com/community/vmtn/vsan

VMware PowerCLI:
https://code.vmware.com/web/dp/tool/vmware-powercli/

VMware API Explorer:
https://code.vmware.com/apis

Support Knowledge base:
https://kb.vmware.com/

VMware Contact Information

For additional information or to purchase VMware Virtual SAN, VMware's global network of solutions providers is ready to assist. If you would like to contact VMware directly, you can reach a sales representative at 1-877-4VMWARE (650-475-5000 outside North America) or email sales@vmware.com. When emailing, please include the state, country, and company name from which you are inquiring.

About the Author

This cookbook was put together using content from various resources from Virtual SAN Engineering.

Jase McCarty is a Staff Technical Marketing Architect at VMware with a focus on storage solutions. He has been in the Information Technology field for over 25 years, with roles on both the customer and vendor side. Jase has Co-Authored two books on VMware virtualization, routinely speaks technology focused user group meetings, and has presented at several industry conferences including VMworld.

Follow Jase on Twitter: @jasemccarty

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com