Capturing installed Windows Features from Client OS into SCCM without touching the MOF

Recently I was asked how to capture the Enabled Windows features from client machines, to help identify the crazy cats who have Hyper-V turned on just so they can run an extra OS or 2 on their Surface Pro 3. So this one was an interesting question in the server OS there is the Win32_ServerFeatures WMI class which is great on servers, but doesn't exist on client OS's. All of the information is captured in DSIM right, but how do you turn that into a WMI Class for reporting purposes.

I started off by creating a PowerShell Script with plans to run it in DCM to create the Registry keys which we have traditionally used to capture the information in SCCM, which looks like this:

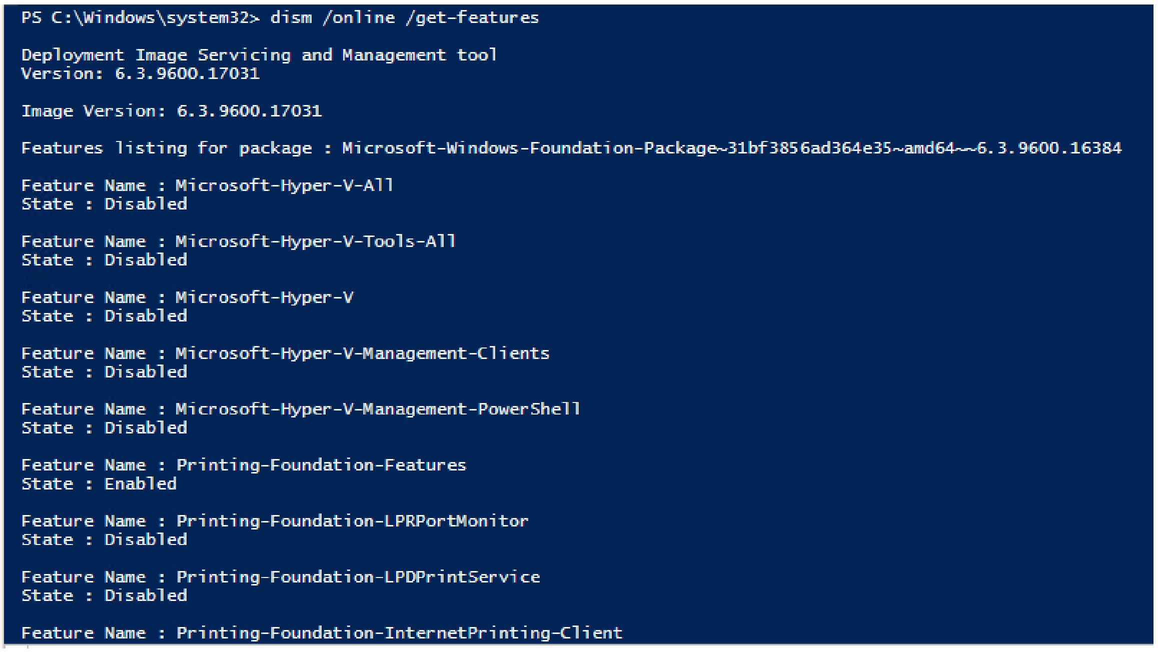
```
$f = ""
$feature = dism /online /get-features
foreach ($svc in $feature)
{
    if ($svc -like "*Feature Name *")
        {
            $f = $f + $svc.Replace("Feature Name : ", "") + ","
        }
    elseif ($svc -like "*State : *")
        {
            $f = $f + $svc.Replace("State : ","") + ";"
        }
}

Push-Location
Set-Location HKLM:
if ((Test-Path .\software\clientFeatures) -eq $false){New-Item -Path .\software -name
clientFeatures | out-null}
foreach ($r in $f.Split(";"))
{
    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        New-ItemProperty .\software\clientFeatures -Name $feat -Value $featset -
PropertyType "string" -Force | Out-Null
        $feat = ""
        $featset = ""
    }
}
Pop-Location
```

So let's breakdown the script and explain what each of the sections are doing

```
$feature = dism /online /get-features
```

This puts the default DISM result into the $feature variable



```
PS C:\Windows\system32> dism /online /get-features

Deployment Image Servicing and Management tool
Version: 6.3.9600.17031

Image Version: 6.3.9600.17031

Features listing for package : Microsoft-Windows-Foundation-Package~31bf3856ad364e35~amd64~~6.3.9600.16384

Feature Name : Microsoft-Hyper-V-All
State : Disabled

Feature Name : Microsoft-Hyper-V-Tools-All
State : Disabled

Feature Name : Microsoft-Hyper-V
State : Disabled

Feature Name : Microsoft-Hyper-V-Management-Clients
State : Disabled

Feature Name : Microsoft-Hyper-V-Management-PowerShell
State : Disabled

Feature Name : Printing-Foundation-Features
State : Enabled

Feature Name : Printing-Foundation-LPRPortMonitor
State : Disabled

Feature Name : Printing-Foundation-LPDPrintService
State : Disabled

Feature Name : Printing-Foundation-InternetPrinting-Client
```

I know your sitting there like me and wondering how you can turn that into a registry entry.

Well that's where the next phase of the script comes in and creates a single string

```
foreach ($svc in $feature)
{
    if ($svc -like "*Feature Name *")
        {
            $f = $f + $svc.Replace("Feature Name : ", "") + ","
        }
    elseif ($svc -like "*State : *")
        {
            $f = $f + $svc.Replace("State : ","") + ";"
        }
}
```

Which steps through each of the lines in the result of the DISM command, and finds the lines which have "Feature Names" and "State" then writes into string called $f which we will use later. As we have put the , after the Feature name and the ; after the state, we can then split these in the next phase of the script. Which I will now explain what we are doing there:

```
Push-Location
Set-Location HKLM:
if ((Test-Path .\software\clientFeatures) -eq $false){New-Item -Path .\software -name
clientFeatures | out-null}
foreach ($r in $f.Split(";"))
{
    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        New-ItemProperty .\software\clientFeatures -Name $feat -Value $featset -
PropertyType "string" -Force | Out-Null
        $feat = ""
        $featset = ""
    }
}
Pop-Location
```

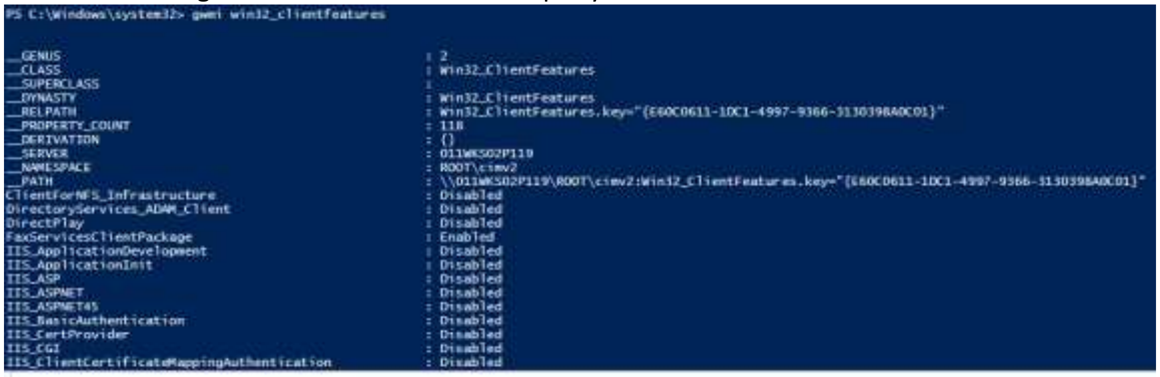
At a high level we are connecting to HKEY_LOCAL_MACHINE and created a new Key called ClientFeatures under the Software key, then creating a new REG_SZ for each of the features with the states as the value of the key. At this point I was really happy I had all of the features in the registry in a format I could then create a MOF from, this was short lived as I noticed I had over 100 features in the client OS which would create a heartache when creating the MOF file as it would be around 300 lines of boring code. So I went to myself, self why don't you just create the WMI class with PowerShell then you don't have to worry about getting the MOF file right, which I responded to self with that's a great idea I'm amazed I didn't think of it myself. Below you will see the resulting script:

```
if ((Get-WmiObject -Class Win32_ClientFeatures -ErrorAction
SilentlyContinue).__PROPERTY_COUNT -lt 1)
{
    $newClass = New-Object System.Management.ManagementClass("root\cimv2",
[String]::Empty, $null)
    $newClass["__CLASS"] = "Win32_ClientFeatures"
    $newClass.Qualifiers.Add("Static", $true)
    $newClass.Properties.Add("key",[System.Management.CimType]::String, $false)
    $newClass.Properties["key"].Qualifiers.Add("Key", $true)
    foreach ($r in $f.Split(";"))
    {
        if ($r -ne "")
        {
            $feat = $r.Split(",")[0].ToString()
            $featset = $r.Split(",")[1].ToString()
            $newClass.Properties.Add($feat,[System.Management.CimType]::String,
$false)
            $newClass.Put() | Out-Null
            $feat = ""
            $featset = ""
        }
    }
}
if ((Get-WmiObject -Class Win32_ClientFeatures).__path -eq $null){$new = $true} else
{$inst = (Get-WmiObject -Class Win32_ClientFeatures).key}
if ($new) {$classinstance = $newClass.CreateInstance()}
foreach ($r in $f.Split(";"))
{
    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        if ($new)
        {
            $classinstance.$feat = $featset
            $classinstance.put() | Out-Null
        }
        else
        {
            $fe = Get-WmiObject -Class Win32_clientFeatures
            $fe.$feat = $featset
            $fe.put() | Out-Null
        }
        $feat = ""
        $featset = ""
    }
}
```

Let's dive into the breakdown of the slab of code,

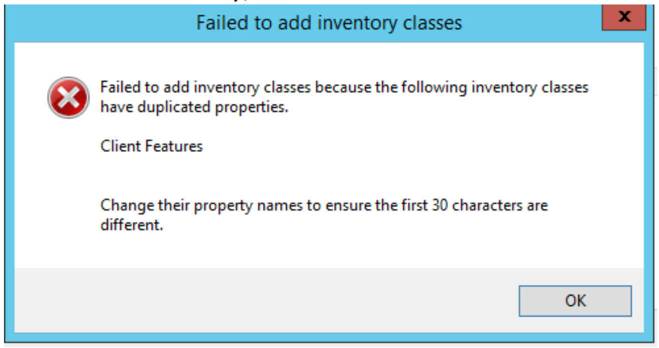
```
if ((Get-WmiObject -Class Win32_ClientFeatures -ErrorAction
SilentlyContinue).__PROPERTY_COUNT -lt 1)
```

This if statement is checking to see if the WMI Class called Win32_ClientFeatures has any properties assigned to it, if the count is less than 1 then we will go ahead and create the WMI Class and create a Property for each of the Windows Features which is this slab of code:

```
{
    $newClass = New-Object System.Management.ManagementClass("root\cimv2",
[String]::Empty, $null)
    $newClass["__CLASS"] = "Win32_ClientFeatures"
    $newClass.Qualifiers.Add("Static", $true)
    $newClass.Properties.Add("key",[System.Management.CimType]::String, $false)
    $newClass.Properties["key"].Qualifiers.Add("Key", $true)
    foreach ($r in $f.Split(";"))
    {
        if ($r -ne "")
        {
            $feat = $r.Split(",")[0].ToString()
            $featset = $r.Split(",")[1].ToString()
            $newClass.Properties.Add($feat,[System.Management.CimType]::String,
$false)
            $newClass.Put() | Out-Null
            $feat = ""
            $featset = ""
        }
    }
}
```
We have also created a key called "key" inventive I know, but it works. So we now have a WMI Class with all of the properties the next step is to create a WMI instance of the state of each of the Windows Features like so:
```
if ((Get-WmiObject -Class Win32_ClientFeatures).__path -eq $null){$new = $true} else
{$inst = (Get-WmiObject -Class Win32_ClientFeatures).key}
if ($new) {$classinstance = $newClass.CreateInstance()}
foreach ($r in $f.Split(";"))
{
    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        if ($new)
        {
            $classinstance.$feat = $featset
            $classinstance.put() | Out-Null
        }
        else
        {
            $fe = Get-WmiObject -Class Win32_clientFeatures
            $fe.$feat = $featset
            $fe.put() | Out-Null
        }
        $feat = ""
        $featset = ""
    }
}
```
The if statement is checking to see if the there is an existing instance in the WMI object and selects that instance to update, otherwise we will go ahead and create a new instance. As you can see in the script these are 2 different groups of code.
So I now have a great WMI class which I can query with PowerShell and it returns like so:

```
PS C:\Windows\system32> gwmi win32_clientfeatures


__GENUS                                       : 2
__CLASS                                       : Win32_ClientFeatures
__SUPERCLASS                                  :
__DYNASTY                                     : Win32_ClientFeatures
__RELPATH                                     : Win32_ClientFeatures.key="{E60C0611-1DC1-4997-9366-313039BA0C01}"
__PROPERTY_COUNT                              : 118
__DERIVATION                                  : {}
__SERVER                                      : 011WKS02P119
__NAMESPACE                                   : ROOT\cimv2
__PATH                                        : \\011WKS02P119\ROOT\cimv2:Win32_ClientFeatures.key="{E60C0611-1DC1-4997-9366-313039BA0C01}"
ClientForNFS_Infrastructure                   : Disabled
DirectoryServices_ADAM_Client                 : Disabled
DirectPlay                                    : Disabled
FaxServicesClientPackage                      : Enabled
IIS_ApplicationDevelopment                    : Disabled
IIS_ApplicationInit                           : Disabled
IIS_ASP                                       : Disabled
IIS_ASPNET                                    : Disabled
IIS_ASPNET45                                  : Disabled
IIS_BasicAuthentication                       : Disabled
IIS_CertProvider                              : Disabled
IIS_CGI                                       : Disabled
IIS_ClientCertificateMappingAuthentication    : Disabled
```

We are now sitting there really happy with our new WMI class which captures all of the Windows Features settings for the computer great. The next step is to import the new WMI class into SCCM Hardware Inventory, and this is where it became interesting, as I got this error message:


Failed to add inventory classes

Failed to add inventory classes because the following inventory classes have duplicated properties.

Client Features

Change their property names to ensure the first 30 characters are different.

I started doing some digging into this and found that the Remote Server Administration Tools features all started the same along with the language packs, so I tweaked the script to handle this and is still didn't correct the issue. At about this point I came to the realisation that as easy as it would be from a reporting point of view I would be next to impossible to maintain as each OS version has different features, and then you need to include the RSAT features, and Language packs, which would result of in hundreds of properties required in WMI for each OS just to capture the information.

To get around this we then need to change the WMI class to have 2 properties called Feature which we will make as the Key for the instances, and value which will contain the state for the Feature. To complete this we run the following script:

```
$f = ""
$new = $true
$feature = dism /online /get-features
foreach ($svc in $feature)
{
    if ($svc -like "*Feature Name *")
    {
        $f = $f + $svc.Replace("Feature Name : ", "") + ","
    }
    elseif ($svc -like "*State : *")
    {
        $f = $f + $svc.Replace("State : ","") + ";"
    }
}
if ((Get-WmiObject -Class ClientFeatures -ErrorAction
SilentlyContinue).__PROPERTY_COUNT -lt 1)
{
    $newClass = New-Object System.Management.ManagementClass("root\cimv2",
[String]::Empty, $null)
    $newClass["__CLASS"] = "Win32_ClientFeatures"
    $newClass.Qualifiers.Add("Static", $true)
    $newClass.Properties.Add("Feature",[System.Management.CimType]::String, $false)
    $newClass.Properties["Feature"].Qualifiers.Add("Key", $true)
    $newClass.Properties.Add("value",[System.Management.CimType]::String, $false)
    $newClass.Put() | out-null
}

foreach ($r in $f.Split(";"))
{

    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        if ((Get-WmiObject -Class Win32_ClientFeatures).feature -eq $feat)
        {
            $fe = Get-WmiObject -query "select * from win32_clientFeatures where
feature = '$feat'"
            $fe.value = $featset
            $fe.put() | Out-Null
        }
        else
        {
            $classinstance = $newClass.CreateInstance()
            $classinstance.feature = $feat
            $classinstance.value = $featset
            $classinstance.put() | Out-Null
        }
        $feat = ""
        $featset = ""
    }
}
```

The first part is the same as above, but once we get to defining the WMI class it changes a little, now rather than stepping through each of windows features and creating a property for each feature we are now just creating 2 properties called "Feature" and "Value", only if the WMI class doesn't exist, as defined in the below script:

```
if ((Get-WmiObject -Class ClientFeatures -ErrorAction
SilentlyContinue).__PROPERTY_COUNT -lt 1)
{
    $newClass = New-Object System.Management.ManagementClass("root\cimv2",
[String]::Empty, $null)
    $newClass["__CLASS"] = "Win32_ClientFeatures"
    $newClass.Qualifiers.Add("Static", $true)
    $newClass.Properties.Add("Feature",[System.Management.CimType]::String, $false)
    $newClass.Properties["Feature"].Qualifiers.Add("Key", $true)
    $newClass.Properties.Add("value",[System.Management.CimType]::String, $false)
    $newClass.Put() | out-null
}
```

The next step is to create a new instance for each features, defining the feature name and the state like so:

```
foreach ($r in $f.Split(";"))
{

    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        if ((Get-WmiObject -Class win32_ClientFeatures).feature -eq $feat)
        {
            $fe = Get-WmiObject -query "select * from win32_clientFeatures where
feature = '$feat'"
            $fe.value = $featset
            $fe.put() | Out-Null
        }
        else
        {
            $classinstance = $newClass.CreateInstance()
            $classinstance.feature = $feat
            $classinstance.value = $featset
            $classinstance.put() | Out-Null
        }
        $feat = ""
        $featset = ""
    }
}
```

We are also checking to see if the instance already exists for the feature, if it does exist, we will need to update the value, otherwise we create a new class instance.

Now when we import the WMI class into the SCCM console it doesn't throw an error, and we can see the v_GS_ClientFeatures in SQL Reporting Services as an option to report upon now, so how do I get the list of computers which have Hyper-V Enabled, well you create a SQL Report with the following query:

```
select rsys.Name0
from v_R_System as rsys join
```

```
v_GS_CLIENT_FEATURES as feat on rsys.ResourceID = feat.ResourceID
where Feature0 = 'Microsoft-Hyper-V' and
value0 = 'enabled'
```

And the WMI Query for a collection looks like this:
select * from SMS_R_System inner join SMS_G_System_CLIENT_FEATURES on SMS_G_System_CLIENT_FEATURES.ResourceId = SMS_R_System.ResourceId where SMS_G_System_CLIENT_FEATURES.Feature = "Microsoft-Hyper-V" and SMS_G_System_CLIENT_FEATURES.value = "Enabled"

The DCM detection PowerShell script looks like this:

```
$compliance = "Compliant"
$f = ""
$feature = dism /online /get-features
foreach ($svc in $feature)
{
    if ($svc -like "*Feature Name *")
    {
        $f = $f + $svc.Replace("Feature Name : ", "") + ","
    }
    elseif ($svc -like "*State : *")
    {
        $f = $f + $svc.Replace("State : ","") + ";"
    }
}
if ((Get-WmiObject -Class win32_ClientFeatures -ErrorAction
SilentlyContinue).__PROPERTY_COUNT -eq 2)
{
    foreach ($r in $f.Split(";"))
    {
        if ($r -ne "")
        {
            $feat = $r.Split(",")[0].ToString()
            $featset = $r.Split(",")[1].ToString()
            $cls = Get-WmiObject -query "select * from win32_clientFeatures where
feature = '$feat'"
            if ($cls.value -ne $featset) {$compliance = "Non-Compliant"}
            $feat = ""
            $featset = ""
        }
    }
}
else
{
    $compliance = "Non-Compliant"
}
$compliance
```

And the DCM remediation PowerShell script looks like this:

```
$f = ""
$new = $true
$feature = dism /online /get-features
foreach ($svc in $feature)
{
    if ($svc -like "*Feature Name *")
    {
        $f = $f + $svc.Replace("Feature Name : ", "") + ","
    }
    elseif ($svc -like "*State : *")
    {
        $f = $f + $svc.Replace("State : ","") + ";"
    }
}
if ((Get-WmiObject -Class ClientFeatures -ErrorAction
SilentlyContinue).__PROPERTY_COUNT -lt 1)
{
    $newClass = New-Object System.Management.ManagementClass("root\cimv2",
[String]::Empty, $null)
    $newClass["__CLASS"] = "Win32_ClientFeatures"
    $newClass.Qualifiers.Add("Static", $true)
    $newClass.Properties.Add("Feature",[System.Management.CimType]::String, $false)
    $newClass.Properties["Feature"].Qualifiers.Add("Key", $true)
    $newClass.Properties.Add("value",[System.Management.CimType]::String, $false)
    $newClass.Put() | out-null
}

foreach ($r in $f.Split(";"))
{

    if ($r -ne "")
    {
        $feat = $r.Split(",")[0].ToString()
        $featset = $r.Split(",")[1].ToString()
        if ((Get-WmiObject -Class Win32_ClientFeatures).feature -eq $feat)
        {
            $fe = Get-WmiObject -query "select * from win32_clientFeatures where
feature = '$feat'"
            $fe.value = $featset
            $fe.put() | Out-Null
        }
        else
        {
            $classinstance = $newClass.CreateInstance()
            $classinstance.feature = $feat
            $classinstance.value = $featset
            $classinstance.put() | Out-Null
        }
        $feat = ""
        $featset = ""
    }
}
$compliance = "Compliant"
$compliance
```

For the DCM you will need to do a detection on a string that equals "Compliant" and remediate if it not.

As an recap of the whole blog, you can see there is a couple of ways to capture the information, depending upon the amount of properties it might make sense to have a single WMI instance in the WMI class, but in other cases it might make sense to have a large number of WMI instances in the WMI class, each have their advantages and disadvantages for the methods. The nice part of this solution is we don't need to worry about making any changes to the MOF which makes life so much easier.

Good Luck

Steve