PowerShell for Microsoft Exchange Message Tracking

As a Microsoft Exchange 2010 administrator I am often asked to get information about messages that have been sent, received, or that didn't get to or from their destination. Microsoft provides the Message Tracking tools to be able to do this from the Exchange Management Console in Exchange 2010.

In Microsoft Exchange 2003 the Message Tracking feature entirely based in the administrative console, but luckily since Exchange 2007 we have been able to access the same features and functions using PowerShell and the Exchange Management Shell.

Let's assume that you have been asked by someone in your client environment who needed to see if an email was received properly. They provide you with the information that you request which will be the sender address (someaddress@yourdomain.com), subject (Important information that I need feedback on) and a time window in which to search (May 25th between 5-8:30 AM).

Using this criteria, we will feed parameters to the Get-MessageTrackingLog CmdLet as follows:

```
get-messagetrackinglog -Sender someaddress@yourdomain.com -MessageSubject "Important information that I need feedback on" -Start "5/25/2012 5:00:00 AM" -End "5/25/2012 8:30:00 AM"
```

No really, it is just that easy. You can get the same results through the Exchange Management Console, but by using the PowerShell environment you can also take the results and perform actions against them such as further filtering, sorting and passing the information to and from other processes or to output files.

This is one if the many very handy CmdLets to have ready for administering your environment. This can be run from the server console or you may also run it from a workstation which has the Exchange Management Shell installed locally. This is how I do most of my Exchange 2010 PowerShell management which reduces the need to be logging onto server consoles unnecessarily.

I hope that this saves you some time, and feel free to experiment with the parameters and using the pipeline features of PowerShell to manage the results for your specific needs. The TechNet article for the Get-MessageTrackingLog CmdLet can be found here:
http://technet.microsoft.com/en-us/library/aa997573.aspx

Happy Scripting!

PowerShell – Copy Exchange 2010 Receive Connectors Between Servers

The situation that many Exchange administrators are in, is a simple one. Multiple CAS servers for redundancy and fault tolerance, as well as load balancing either inside the cluster, or on the outside using something like a Citrix Netscaler or F5 device. This is a natural design for Exchange 2010 and provides great functionality and recovery capability.

There are also many designs which use a single server, or perhaps a single server per site contains Receive Connectors for inbound relay based on a set of criteria such as auth type or network range.

The challenge for Receive Connectors, and especially with network ranges applied, is that they are difficult to re-create on another server. Whether it is on your first creation, or if you want to make a copy on another server for active or passive use, PowerShell can be your best friend for this task.

Assuming that we have two servers named **EXSITE1** and **EXSITE2** where we have created our Receive Connector named ***Default-App-Connector*** on **EXSITE1** but we want to have it on **EXSITE2**. While we can create one easily enough using the **New-ReceiveConnector** CmdLet we have to type in all of the IP ranges manually which is both tedious and error prone.

Here is your solution. The basic command for creating the new backup connector is:

```
New-ReceiveConnector “Default-App-Connector” -Server EXSITE2 -Bindings 0.0.0.0:25
```

The problem is that this only creates the connector, but not the IP ranges. As I mentioned, if we type the allowed IP addresses and ranges into the command using the ***-RemoteIPRanges*** parameter I have a lot of work ahead of me.

So we simply read the ***-RemoteIPRanges*** from the first connector and pass them to the **New-ReceiveConnector** CmdLet just like so:

```
New-ReceiveConnector “Default-App-Connector” -Server EXSITE2 -Bindings 0.0.0.0:25 -RemoteIPRanges ( Get-ReceiveConnector “EXSITE1Default-App-Connector” ).RemoteIPRanges
```

You can also use the **Get-ReceiveConnector** to document your configuration to file, which is a good practice for BCP. Because these can be volatile, I recommend you export to disk, or replicate to the second server weekly or monthly.

Simply use this command:

```
Get-ReceiveConnector “EXSITE1Default-App-Connector” | Format-List | Out-File “X:ExchangeConfigurationDefault-App-Connector.txt”
```

It’s just that easy. A simple command that can provide peace of mind and protection. Hopefully you

find this to be helpful.

Exchange 2010 SP2 and Citrix Netscalers – SSL gotcha

This is a quick note about a real issue that I ran into very recently. With the Microsoft Exchange 2010 SP2 update there are a number of things to be careful of. With any of the major rollups and service packs, there are often default configuration settings which are re-enabled as a part of the update.

If you've been running Exchange with Outlook Web Access then you have most likely hit the first of two issues which is that any customized OWA pages (logon, logoff etc.) are overwritten during updates. Make sure that you save any custom configurations, and if you have configured a different default theme to be used by your CAS servers in 2010, you may also have to reset the default again to your chosen theme.

The second issue which will really twist you up is if you are using a Network Load Balancer (NLB) such as the Citrix Netscaler appliance. The reason that this is an issue is because of the SSL offload capability which is one of the great advantages of these devices.

During the update of the CAS roles to SP2, the Default Web Site suddenly disappeared from the active monitoring despite the fact that the site itself was still up when accessed directly from the local server using the https://yourservername/owa URL. The key to this is that during the update the SSL option was re-enabled on the Default Web Site for the CAS server.

Because we use the NLB for managing the SSL we can safely uncheck the SSL Required option within IIS and as if by magic, the site is now available again through the load balanced configuration.

Importing, and Updating Exchange 2010 Contacts from CSV

I've had a very specific requirement to add a list of external contacts into an internal Microsoft Exchange environment. The goal of this is to integrate an external company's list of contacts and

some key information into the Global Address List with an identifier on each record showing the company name as part of the display name.

Importing and Updating Exchange Contacts from CSV

Before we go further, I want to qualify the process and subsequent script which supports it. This article, and the script is designed towards the intermediate to advanced PowerShell admin. By this I mean that I haven't exhaustively documented all of the commands and steps. The article is presented in full length so I apologize for the long read.

The requirements for the script are as follows:

- The contacts must appear in the Global Address List along with existing mail accounts
- Display names contain the company name as a suffix to differentiate from employees
- Updates include a number of fields such as address, title, phone number, name
- Source system uses legal name and informal name. We want to display the informal name (i.e. Chuck vs. Charles)
- Updates will be done programatically and fairly frequently (let's assume weekly)
- Updates cannot impact existing mailflow (Must update rather than delete and re-create)
- The script was designed to run using the Microsoft Exchange 2010 Command Shell

The file that we receive will be in CSV format and has a header row. This is a saving grace as we can easily manage the content easily using the column header inside a ForEach loop.

THE FILE

The columns that we receive and a description is shown below:

EMail (SMTP formatted address)
InformalName (If FirstName is different than the name used commonly)
FirstName (Legal first name)
Lastname (Last name)
BranchCode (Each branch is assigned a code in our example)
BranchLegalName (Branch description)
AddressLine1 (Address information)
AddressLine2 (Address information if needed)
AddressCity (City name)
AddressState (State or province)
AddressZip (ZIP or Postal Code)
AddressCountry (Country in ISO code format)
JobTitle (Job title)
PhoneCountryCode (If exists for non-North American numbers)
PhoneAreaCode (Area code)
PhoneNumber (Phone number in ###-#### format)
PhoneExtensionOrSpeedDial (Extension or speed dial number if it exists)

That's a lot of information and because it comes from an external system, the fields don't match up one-to-one with the native Microsoft Exchange fields which are associated with a mail contact.

THE DESIGN

Now we get to the good stuff! Here is the logic of how I built the script that is used:

1. Read the file
2. Loop through the contacts in the file
3. Check to see if the contact exists
4. If it doesn't exists, create it
5. If the contact exists, update it
6. Loop ends
7. Read the OU where we store the contacts
8. Read the file
9. Loop through the contacts in AD
10. Check to see if the contact exists in the file
11. If it doesn't exist, delete it
12. If it exists, do nothing
13. Loop ends

Thanks to the magic of PowerShell this is a fairly simple task. We will use the following CmdLets to accomplish our task:

- Import-CSV
- ForEach
- If, Else, ElseIf
- Get-Contact (Exchange 2010)
- Set-MailContact (Exchange 2010)
- Set-Contact (Exchange 2010)
- New-MailContact (Exchange 2010)
- Get-Content
- Write-Host

I've used the Write-Host CmdLet to output to screen so that we can troubleshoot and monitor the process during the initial tests. Another important feature we use for testing is the -WhatIf parameter. I'll give the necessary disclaimer here which is that you must run this in a test environment and using the WhatIf parameter first! It's not that I don't know that the script works, but regardless of my confidence, it is an absolute must that you test any process in an alternate environment before you go live.

THE SCRIPT

Hold on to your hats because this is a big one. We will step through the script together in sections to show what's happening along the way.

```
# Set the file location
$rawfile = "X:SCRIPTScontacts.csv"

# Ignore Error Messages and continue on. Comment this out with a # symbol if you
need verbose errors
trap [Microsoft.Exchange.Configuration.Tasks.ManagementObjectNotFoundException] {
continue; }
```

```
$contactfile = Import-CSV $rawfile
```

In this section we've identified the location of the import file (assume **X:SCRIPTS** for the path and a filename of **contacts.csv**), as well as setting the alerts to continue on error. Note that this is not alwasy 100% effective and may require some tweaking which I'll update as I make more progress with the error handling. Next we see that the file is imported into the $contactfile as an array.

Next up we will loop through the records and assign working variables to based on the contents of each record. For the name fields there have been some issues where people have names containing spaces (e.g. FirstName=Mary Jane) which will cause the import to fail. For these cases we will use a **-replace** option when assigning the value to the variable and replace the spaces with hyphens.

```
# Read contacts and make the magic happen
ForEach ($contact in $contactfile) {

# Read all attributes provided by the file
$sourceEMail=$contact.EMail
$sourceInformalName=$contact.InformalName -replace " ","-"
$sourceFirstName=$contact.FirstName -replace " ","-"
$sourceLastName=$contact.LastName -replace " ","-"
$sourceManagerID=$contact.ManagerID
$sourceBranchCode=$contact.BranchCode
$sourceBranchLegalName=$contact.BranchLegalName
$sourceAddressLine1=$contact.AddressLine1
$sourceAddressLine2=$contact.AddressLine2
$sourceAddressLine3=$contact.AddressLine3
$sourceAddressCity=$contact.AddressCity
$sourceAddressState=$contact.AddressState
$sourceAddressZip=$contact.AddressZip
$sourceAddressCountry=$contact.AddressCountry
$sourceJobTitle=$contact.JobTitle
$sourcePhoneCountryCode=$contact.PhoneCountryCode
$sourcePhoneAreaCode=$contact.PhoneAreaCode
$sourcePhoneNumber=$contact.PhoneNumber
$sourcePhoneExtensionOrSpeedDial=$contact.PhoneExtensionOrSpeedDial
```

Now we will take the variables which have been assigned and begin to manipulate the data into fields where we require concatenation of the data. This is also where we do the logical checks for the InformalName field by checking the length of the field. If the field is greater than a zero length then it will be used for the First Name attribute on the contact.

We also craft the Display Name by concatenating the calculated First Name, the Last Name and the trailing suffix of COMPANY (you can change that to be whatever identifier you want). Address and phone number are fairly simple, but again we check for field values of zero length to decide if we need to include them in the concatenated results.

Lastly in this section we create the Alias which must be unique. We prefix with COMPANY again to ensure they are different than our existing user records and make them easily searchable.

```
 # Create the concatenated fields and custom fields

# Informal Name – This checks to see if they have an informal name (Jim versus James)
and if so, use the informal name
if ($sourceInformalName.Length -lt 1) {
$targetFirstName = $sourceFirstName
}
elseif ($sourceInformalName.Length -gt 1) {
$targetFirstName = $sourceInformalName
}

# Assign the Display Name using the correct First Name, Last Name and a suffix of
COMPANY. We trim this field because of leading spaces that show up regularly
$sourceDisplayName = "$targetFirstName $sourceLastName COMPANY"
$targetDisplayName = $sourceDisplayName.Trim()

# Assign the Distinguished Name attribute using the correct First Name, Last Name and
OU structure
$targetDistinguishedName = "CN=$targetFirstName
$sourceLastName,OU=ExternalContacts,DC=yourdomain,DC=com"

# Assemble the phone number

# Check for a country code, otherwise value is null
if ($sourcePhoneCountryCode -lt 1) {
$targetCountryCode = $null
}
elseif ($sourcePhoneCountryCode -gt 1) {
$targetCountryCode = "$sourceCountryCode-"
}

# Check for an extension, otherwise value is null
if ($sourcePhoneExtensionOrSpeedDial -lt 1) {
$targetExtension = $null
}
elseif ($sourcePhoneExtensionOrSpeedDial -gt 1) {
$targetExtension = " ext. $sourcePhoneExtensionOrSpeedDial"
}

$targetPhoneNumber = "$targetCountryCode$sourcePhoneAreaCode-
$sourcePhoneNumber$targetExtension"
# Assemble the Address
$targetStreetAddress = "$sourceAddressLine1 $sourceAddressLine2
$sourceAddressLine3"

# Assign the name attribute for new contacts
$targetCommonName = "$sourceFirstName $sourceLastName"

# Assign the Alias using COMPANY as a prefix so that we can identify them easily
$targetAlias = "COMPANY$targetFirstName$sourceLastName"
```

So what we have got now is a working set of data to begin to apply to our Exchange environment.

The next step is to search Active Directory/Exchange for the contact to see if they are existing.

```
#####################################################
# Search for the contact to see if it is existing #
#####################################################

if ( Get-Contact -Identity $sourceEmail )
{
# Output to screen so we can track the process. Comment the following line when it is
running as a batch process
Write-Host $sourceEmail Exists so $targetDisplayName will be MODIFIED -
foregroundcolor green

Set-MailContact -Identity $sourceEmail -Alias $targetAlias -ForceUpgrade
Set-Contact -Identity $sourceEmail `
-City $sourceAddressCity `
-Company $sourceBranchLegalName `
-CountryOrRegion $sourceAddressCountry `
-Department $sourceBranchCode `
-DisplayName $targetDisplayName `
-SimpleDisplayName $targetDisplayName `
-Name "$targetCommonName" `
-FirstName $targetFirstName `
-LastName $sourceLastName `
-Phone $targetPhoneNumber `
-PostalCode $sourceAddressZip `
-StateOrProvince $sourceAddressState `
-StreetAddress $targetStreetAddress `
-Title $sourceJobTitle `
-WebPage "RJFAccountFlag" `
-WindowsEmailAddress $sourceEmail -WhatIf
}
```

Notice that we have done 2 important things in the Set-Contact CmdLet phrasing. For readability we use the ` character which allows you to span multiple lines in a single command. Be careful that you note that it is the reverse single quote (found on the tilde ~ button) and not the traditional single quote ' found by the Enter Key. Secondly we have tagged the command with the -WhatIf parameter to monitor the potential result.

```
#####################################################
# If it is not existing, create a new contact     #
#####################################################
else
{
# Output to screen so we can track the process. Comment the following line when it is
running as a batch process
Write-Host $sourceEmail Does Not Exist so $targetDisplayName will be CREATED -
foregroundcolor yellow
```

```
# First we create the contact with the required properties
New-MailContact -Name "$targetCommonName" `
-OrganizationalUnit "OU=ExternalContacts,DC=yourdomain,DC=com" `
-ExternalEmailAddress $sourceEmail `
-Alias $targetAlias `
-DisplayName $targetDisplayName `
-FirstName $targetFirstName `
-LastName $sourceLastName `
-PrimarySmtpAddress $sourceEmail -WhatIf

# Now we set the additional properties that aren't accessible by the New-MailContact
cmdlet
Set-Contact  -Identity $sourceEmail `
-City $sourceAddressCity `
-Company $sourceBranchLegalName `
-CountryOrRegion $sourceAddressCountry `
-Department $sourceBranchCode `
-DisplayName $targetDisplayName `
-SimpleDisplayName $targetDisplayName `
-FirstName $targetFirstName `
-LastName $sourceLastName `
-Phone $targetPhoneNumber `
-PostalCode $sourceAddressZip `
-StateOrProvince $sourceAddressState `
-StreetAddress $targetStreetAddress `
-Title $sourceJobTitle `
-WebPage "RJFAccountFlag" `
-WindowsEmailAddress $sourceEmail -WhatIf
}

# Clean up after your pet – this does some memory cleanup
[System.GC]::Collect()
[System.GC]::WaitForPendingFinalizers()
}
```

For a new contact we had to perform two steps. The first step is to create the contact with the required attributes. There are limited attributes that can be affected with the **New-MailContact** CmdLet unfortunately, so we follow that command with a **Set-Contact** to update the remaining attributes that we need.

The section I've named as "Clean up after your pet" is a little memory cleansing process. Let's just say that if you skip this step you will find yourself about 4GB deep in memory usage and crawling your way through the contact script.

The last portion of our script is the deletion process. Let me reiterate the importance of testing on this. If there is an error with the reading of the file, or you use a sample file with less contacts it will relentlessly delete them while you watch in horror. The **-WhatIf** has been put into the command here also to help with the assurance testing.

```
##################################################
```

```
############
# Now we reverse the process and remove contacts no longer valid #
# If contact is not in the file, delete it                       #
##################################################################
############

$CurrentContacts = Get-MailContact -OrganizationalUnit
'OU=ExternalContacts,DC=yourdomain,DC=com' -ResultSize Unlimited | Select-Object
PrimarySMTPAddress,name
ForEach ($contact in $currentContacts) {
$sourceEmail = $Contact.PrimarySMTPAddress
if ( Get-Content $rawFile | Select-String -pattern $sourceEmail )
{
# Output to screen so we can track the process. Comment the following line when it is
running as a batch process
Write-Host This contact $sourceEmail Exists in the file -foregroundcolor green
}
else
{
# Output to screen so we can track the process. Comment the following line when it is
running as a batch process
Write-Host "DELETE THIS CONTACT $sourceEmail" -backgroundcolor yellow
Remove-MailContact -Identity "$sourceEmail" -Confirm:$Y -WhatIf
}
}

$CurrentContacts = $null
# Clean up after your pet – this does some memory cleanup
[System.GC]::Collect()
[System.GC]::WaitForPendingFinalizers()
```

Congratulations! We've made it to the end. This script shows you the power and flexibility of PowerShell. For your own situation you can work from this sample and hopefully it will save you some development time.

This is quite a process to go through, so I encourage you to read through it and feel free to add comments with any questions.

Here are the files associated with this process:

Contacts.csv (sample contact file)

ManageExternalContacts.ps1.txt (rename to .ps1 after download)