Chapter 8 - Miscellaneous

In This Chapter

- Introduction
- Menus
- Aliases
- Foreach-Object (%)
- PowerShell Interface Customization
- Command Logging

Introduction

In this book, we've covered numerous topics from how to start out scripting, to customizing Exchange Online topics in PowerShell. The topics picked are, as with the rest of the book, based on practical experience and are ones that should prove useful in a production environment.

For this chapter, we'll cover menus, aliases, Foreach-Object filtering, and special permission cmdlets. Each of these provide some added benefit to managing Exchange Online with PowerShell. Aliases provide a way to customize PowerShell for easier coding. These shortcuts simply make coding easier. In addition to this, the shell can also be customized in terms of path, colors and window sizing.

An additional filtering option will also be covered in this chapter. This cmdlet helps sort through or manipulate results of cmdlets and can be used to provide an 'in-flight' cleanup for results for readability. Foreach-Object is indeed a useful cmdlet for your PowerShell scripts or one-liners.

Menus

Building menus is not a task that is necessary for one off scripts. Menus should be used on scripts that will be run on multiple occasions, for example supporting a large number of mailboxes in Office 365 or for occasions where occasional input may be necessary. Another reason to use it is for a reusable script, which is especially useful for consultants who run their scripts in dozens of environments a year. The menu simply makes running the script quicker and more flexible.

In a PowerShell script, the menu can consist of two parts. The first part is the text for the menu which is the visual part of the script. The menu can be simple and singular in color or very colorful like the example given in Chapter 2. The second part is the infrastructure or back-end of the menu itself. This is where the executable code is stored and where coding needs to be performed in the form of functions that will complete the tasks the menu has called.

Example - Menu 1

```
$Menu = {
Write-Host "******************************************************************"
Write-Host "Office 365 Mailbox Management"
Write-Host "******************************************************************"
Write-Host "1) UPN Check "
Write-Host "2) Configure Retention Policy"
Write-Host "3) Configure Client Access"
Write-Host "4) Set Legal Hold"
Write-Host ""
Write-Host "99) Exit"
Write-Host ""
Write-Host "Select an option.. [1-99]?"
}
```

The above menu has been snipped from a script that is used to manage mailboxes and their settings in Office 365. By itself this menu is just a variable that holds a bunch of text that looks like a menu. Next, we need to build the backbone on the infrastructure part of the menu. This is where PowerShell will make calls to functions in the rest of the script to perform the functions you code for.

To start this section of code, construct a ‘Do While’ code block. The reason for this is that script will keep running options and displaying the menu until an exit code is chosen. So the ‘Do While’ block would look something like this:

```
Do {
Invoke-Command -ScriptBlock $Menu
$Choice = Read-Host
} While ($Choice -ne 99)
```

Notice that with this code, the loop will keep displaying the menu after each option is chosen until the value of 99 is selected. At that point the script will stop and exit to a PowerShell prompt. The Read-Host will store the value type in $opt to be used for selecting which code block to run. Next, there needs to be a way to decide which option will run. What PowerShell cmdlet will allow for this?

```
Switch ($Choice)
```

However, a review of the help on 'Switch' does not reveal a lot of clues for its usefulness/functionality. However, with a little bit of help from your favorite search engine, one can find this MSDN link for PowerShell functionality:

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch?view=powershell-5.1

We find that Switch will act like a condition tester, if a condition is fed to it, it will select that option within the Switch code section. For example:

```
$Choice = 3
Switch ($Choice) {
    1 {
        Write-Host "Thanks for choosing option one."
    }
    2 {
        Write-Host "Thanks for choosing option two."
    }
    3 {
        Write-Host "Thanks for choosing option three."
    }
}
```

The result of this will display the results from choosing option three:

```
Thanks for choosing option three.
```

Let's incorporate this into our menu infrastructure. Using the above as an example, we'll need to build code blocks for each function we'll need to call from our example on the previous page. To make this process simpler (in terms of the scope of the menu) functions are pre-created:

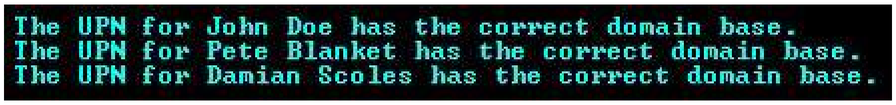
- **Correct mailbox UPN**: Modifies a user's UPN for Exchange Online
- **Configure Retention Policy**: Applies a retention policy to a mailbox
- **Configure Client Access**: Configures mailbox access to client protocols like ActiveSync
- **Set Legal Hold**: Configures a hold on a mailbox in Exchange Online

With these functions created they can be referred to in each option code block. Note that it is assumed that a connection to the MSOL Service has already been made prior to running any of the options.

Option 1

This option calls the Mailbox UPN check function:

```
Function UPN Check {
$BaseDomain = '@OnlineExchangeBook.onmicrosoft.com'
$Users = Get-MSOLUsers -All
Foreach ($User in $Users) {
$DisplayName = $User.DisplayName
$UPN = (Get-MSOLUser $User).UserPrincipalName
If ($UPN -like $BaseDomain) {
Write-Host 'The UPN for $DisplayName has the correct domain base' -ForegroundColor C\
yan
} Else {
Write-Host 'The UPN for $DisplayName has an incorrect domain base' -ForegroundColor \
Yellow
}
}
}
```

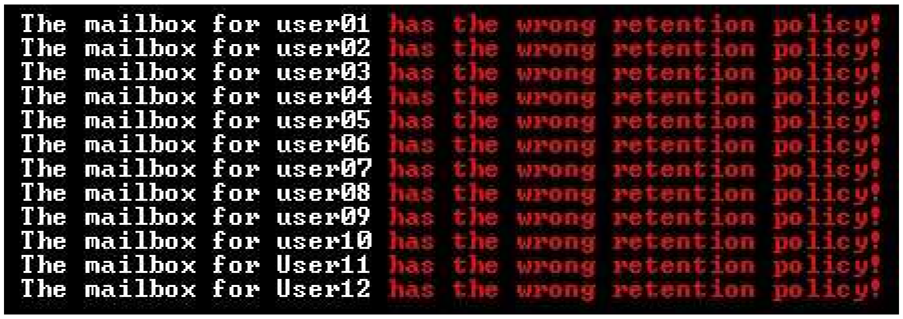


```
The UPN for John Doe has the correct domain base.
The UPN for Pete Blanket has the correct domain base.
The UPN for Damian Scoles has the correct domain base.
```

Option 2

This option calls the Retention Policy Configuration function:

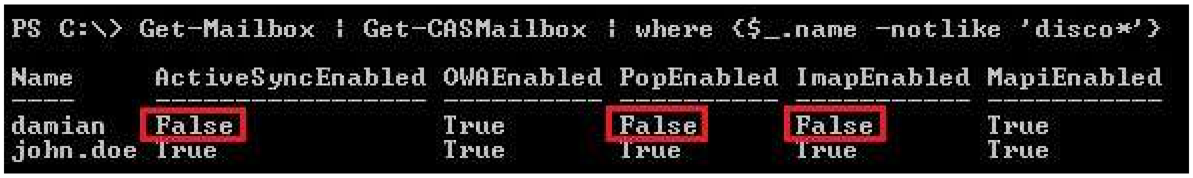
```
Function RetentionPolicy {
$Mailbox = Get-Mailbox
$Policy = '18 Month Email Delete and Purge'
Foreach ($Line in $Mailbox) {
$Name = $Line.DisplayName
$Retention = $Line.RetentionPolicy
If ($Retention -eq $Policy) {
Write-Host "The mailbox for $Name " -ForegroundColor White -NoNewLine
Write-Host "has the right Retention Policy." -ForegroundColor Green
} Else {
Write-Host "The mailbox for $Name " -ForegroundColor White -NoNewLine
Write-Host "has the wrong retention policy!" -ForegroundColor Red
Set-Mailbox $Mailbox -RetentionPolicy $Policy
}
} # End Retention Policy Function
```



```
The mailbox for user01 has the wrong retention policy!
The mailbox for user02 has the wrong retention policy!
The mailbox for user03 has the wrong retention policy!
The mailbox for user04 has the wrong retention policy!
The mailbox for user05 has the wrong retention policy!
The mailbox for user06 has the wrong retention policy!
The mailbox for user07 has the wrong retention policy!
The mailbox for user08 has the wrong retention policy!
The mailbox for user09 has the wrong retention policy!
The mailbox for user10 has the wrong retention policy!
The mailbox for User11 has the wrong retention policy!
The mailbox for User12 has the wrong retention policy!
```

Option 3

This option calls the Configure Client Access function. The reason I want this function is to 'normalize' all access to mailboxes in Exchange Online. Let's say for example the tenant has multiple Global Admins and maybe some mistakes have been made or experimenting with settings. This could potentially lead to problems for end user access. For example, the users below are different, but they should match:

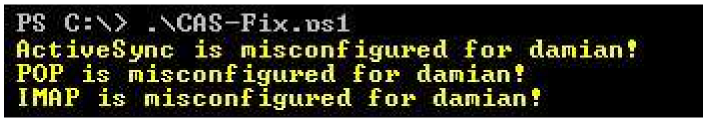


```
PS C:\> Get-Mailbox | Get-CASMailbox | where {$_.name -notlike 'disco*'}

Name     ActiveSyncEnabled OWAEnabled PopEnabled ImapEnabled MapiEnabled
----     ----------------- ---------- ---------- ----------- -----------
damian   False             True       False      False       True
john.doe True              True       True       True        True
```

This function needs to make sure all services are enabled for the user and not disabled like we see above. We will need to get a list of all mailboxes and then use the Set-CASMailbox cmdlet to change the ActiveSync, OWA, POP, IMAP and MAPI settings to be 'Enabled' or set to True ($True in PowerShell). We can add a little feedback if we want to let the admin know what has changed or who was misconfigured if we want to.

```
Function CASChanges {
$Mailboxes = Get-CASMailbox | where {$_.Name -NotLike 'disco*'}
Foreach ($Mailbox in $Mailboxes) {
$Access = Get-CASMailbox | where {$_.Name -NotLike 'Discovery*'}
Foreach ($Line in $Access) {
$Name = $Line.Name
If ($Line.ActiveSyncEnabled -ne $True) {
Write-Host "ActiveSync is misconfigured for $Name!" -ForegroundColor Yellow
Set-CASMailbox $Name -ActiveSyncEnabled $True
}
If ($Line.OWAEnabled -ne $True) {
Write-Host "OWA is misconfigured for $Name!" -ForegroundColor Yellow
Set-CASMailbox $Name -OWAEnabled $True
}
If ($Line.POPEnabled -ne $True) {
Write-Host "POP is misconfigured for $Name!" -ForegroundColor Yellow
Set-CASMailbox $Name -POPEnabled $True
}
If ($Line.IMAPEnable -ne $True) {
Write-Host "IMAP is misconfigured for $Name!" -ForegroundColor Yellow
Set-CASMailbox $Name -IMAPEnabled $True
}
If ($Line.MAPIEnabled -ne $True) {
Write-Host "MAPI is misconfigured for $Name!" -ForegroundColor Yellow
Set-CASMailbox $Name -MAPIEnabled $True
}
}
}
} # End of CAS Changes function
```

Sample run of this function:

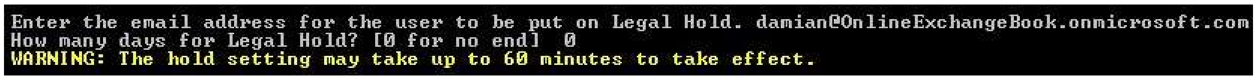


```
PS C:\> .\CAS-Fix.ps1
ActiveSync is misconfigured for damian!
POP is misconfigured for damian!
IMAP is misconfigured for damian!
```

Option 4

The option calls the Legal Hold function:

```
Function LegalHold {
Write-Host 'Enter the email address for the user to be put on Legal Hold. ' -NoNewLi\
ne
$Email = Read-Host
Write-Host 'How many days for Legal Hold? [0 for no end] ' -NoNewLine
$Days = Read-Host
# Set $Days variable to 'Unlimited if a '0' was entered.
If ($Days -eq '0') {
$Days = 'Unlimited'
}
Set-Mailbox $Email -LitigationHoldEnabled $True -LitigationHoldDuration $Days
} # End of Legal Hold function
```



```
Enter the email address for the user to be put on Legal Hold. damian@OnlineExchangeBook.onmicrosoft.com
How many days for Legal Hold? [0 for no end]  0
WARNING: The hold setting may take up to 60 minutes to take effect.
```

Notice that a comment is included in each option block for documentation purposes. For the last option the script provides a way to exit the script cleanly:

Option for Exiting Script

```
99 {# Exit
Write-Host "Exiting..."
}
```

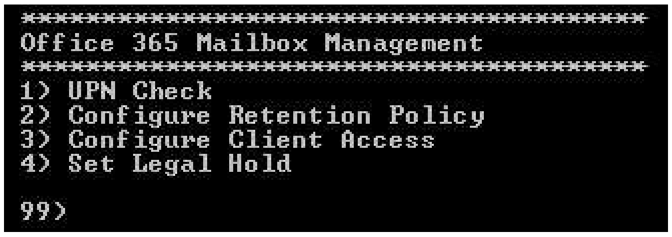
When option 99 is selected, the script exits because of the Do {} While () code block.

Pulling all of the previous code together into one script:

```
$Menu = {
Write-Host "******************************************************************"
Write-Host "Office 365 Mailbox Management"
Write-Host "******************************************************************"
Write-Host "1) UPN Check "
Write-Host "2) Configure Retention Policy"
Write-Host "3) Configure Client Access"
Write-Host "4) Set Legal Hold"
Write-Host "99) Exit"
Write-Host ""
Write-Host "Select an option.. [1-99] " -NoNewLine
}
Do {
Invoke-Command $Menu
$Choice = Read-Host $Menu
```

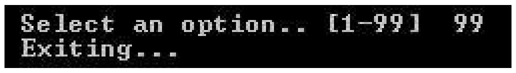
```
Switch ($Choice) {
1 { # UPN Check
UPNCheck
}
2 { # Configure Retention Policy
RetentionPolicy
}
3 { # Configure Client Access
CASChanges
}
4 { # Set Legal Hold
LegalHold
}
99 {# Exit
Write-Host "Exiting..."
}
Default {
Write-Host "You haven't selected any of the available options. "
}
}
} While ($Choice -ne 99)
```

Running the script provides a menu as displayed below:



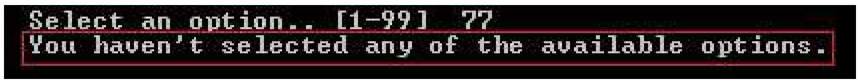
```
****************************************
Office 365 Mailbox Management
****************************************
1) UPN Check
2) Configure Retention Policy
3) Configure Client Access
4) Set Legal Hold

99)
```

Option 99 allows for the script to exit:



```
Select an option.. [1-99]  99
Exiting...
```

If an option is typed in wrong, say 77, an error message is provided (and the script does not exit):

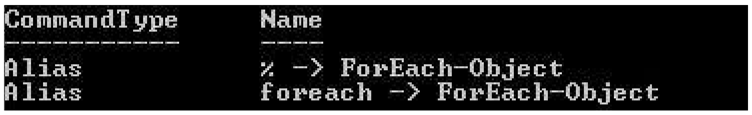


```
Select an option.. [1-99]  77
You haven't selected any of the available options.
```

Aliases

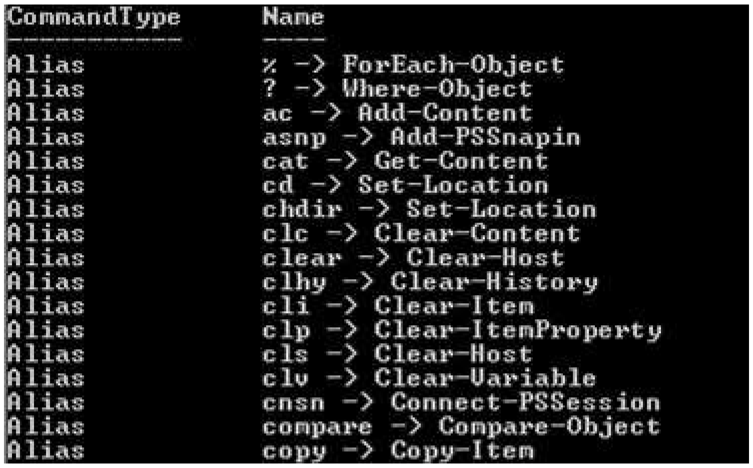
PowerShell aliases are shortened versions of PowerShell cmdlets. Consider aliases to be a convenience in reducing the amount of text in a script. Aliases are not necessary for writing a script but they do provide shortcuts to coding. Without aliases, each command in PowerShell just takes longer to type. The downside of aliases is that normally PowerShell is a very readable scripting language and using aliases can obscure the ability to read PowerShell in plain English. Another downside is that there is no guarantee that the alias will exist in a different environment. If the script is meant to be portable, it would be advisable to not use them or at least limit their usage. If a script will be read by someone other than you, using aliases might make the script unreadable to others.

```
Get-Alias -Definition Foreach-Object
```



```
CommandType     Name
-----------     ----
Alias           % -> ForEach-Object
Alias           foreach -> ForEach-Object
```

However, what if you don't know the command that the alias is for? The above can be reverse engineered to show all aliases. To look up all aliases, simply type in 'Get-Alias':



```
CommandType     Name
-----------     ----
Alias           % -> ForEach-Object
Alias           ? -> Where-Object
Alias           ac -> Add-Content
Alias           asnp -> Add-PSSnapin
Alias           cat -> Get-Content
Alias           cd -> Set-Location
Alias           chdir -> Set-Location
Alias           clc -> Clear-Content
Alias           clear -> Clear-Host
Alias           clhy -> Clear-History
Alias           cli -> Clear-Item
Alias           clp -> Clear-ItemProperty
Alias           cls -> Clear-Host
Alias           clv -> Clear-Variable
Alias           cnsn -> Connect-PSSession
Alias           compare -> Compare-Object
Alias           copy -> Copy-Item
```

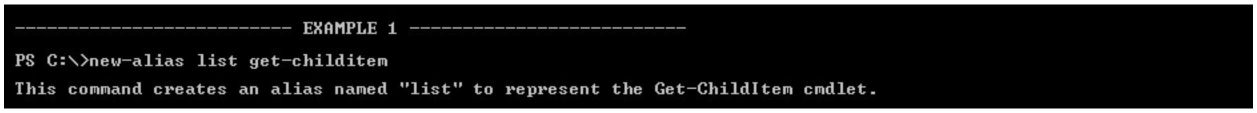
Without listing them all here, all told, there are 148 aliases defined. What may be more interesting is that aliases can be created and modified. This certainly provides for some flexibility or customization of PowerShell.

New-Alias

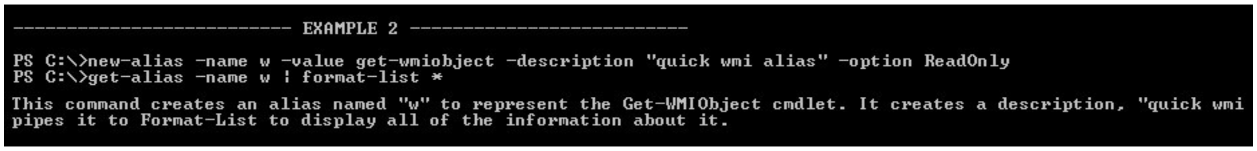
If there is a desire to make custom aliases for PowerShell, New-Alias is the cmdlet to use.

Note: The aliases are only good for the current session. If you close the current session, the alias is lost and when you reconnect to Exchange Online PowerShell the alias will not be there.

```
Get-Help New-Alias -Examples
```



```
-------------------------- EXAMPLE 1 --------------------------
PS C:\>new-alias list get-childitem
This command creates an alias named "list" to represent the Get-ChildItem cmdlet.
```



```
-------------------------- EXAMPLE 2 --------------------------
PS C:\>new-alias -name w -value get-wmiobject -description "quick wmi alias" -option ReadOnly
PS C:\>get-alias -name w | format-list *
This command creates an alias named "w" to represent the Get-WMIObject cmdlet. It creates a description, "quick wmi
pipes it to Format-List to display all of the information about it.
```

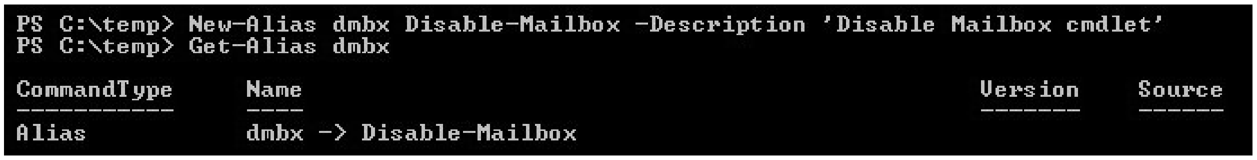
Sample Usage

We can create an alias for just about anything in PowerShell that we want. For this example, we can create aliases for any of the *-Mailbox cmdlets if we wanted to.

To create these aliases, we'll use a series of New-Alias one-liners:

```
New-Alias dmbx Disable-Mailbox -Description 'Disable Mailbox cmdlet'
New-Alias embx Enable-Mailbox -Description 'Enable Mailbox cmdlet'
New-Alias gmbx Get-Mailbox -Description 'Get Mailbox cmdlet'
New-Alias nmbx New-Mailbox -Description 'New Mailbox cmdlet'
New-Alias rmbx Remove-Mailbox -Description 'Remove Mailbox cmdlet'
New-Alias smbx Set-Mailbox -Description 'Set Mailbox cmdlet'
New-Alias gcc Get-ComplianceCase -Description 'Get Compliance Case'
New-Alias ncc New-ComplianceCase -Description 'New Compliance Case'
New-Alias scc Set-ComplianceCase -Description 'Set Compliance Case'
New-Alias rcc Remove-ComplianceCase -Description 'Remove Compliance Case'
```

Example result of a new alias creation:



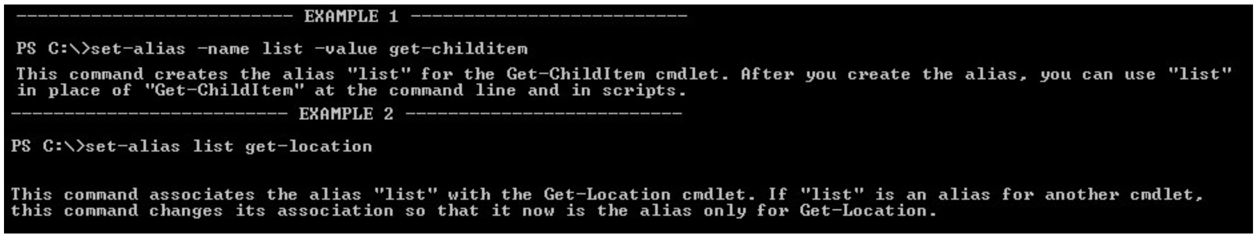
```
PS C:\temp> New-Alias dmbx Disable-Mailbox -Description 'Disable Mailbox cmdlet'
PS C:\temp> Get-Alias dmbx

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dmbx -> Disable-Mailbox
```

There are a few parameters that can be used to customize this new alias during creation. One of the parameters is 'Option' which provides for a way to limit when the alias can be used – Global, Local, Script or Private. An alias could be enabled for only when a script runs or only while in a local session. The purpose of this option is to possibly isolate the usage of a cmdlet as to prevent unwarranted changes using the aliases. A description should be added so that the purpose of the alias is known by others.

Set-Alias

This cmdlet is used to modify any of the existing aliases to the specifics that you may want to configure for a particular alias. One of the exceptions is if the alias is set to ReadOnly. To modify one of those aliases, a '-Force' switch must be used. Here are some sample uses of the cmdlet:

```
Get-Help New-Alias -Examples
```



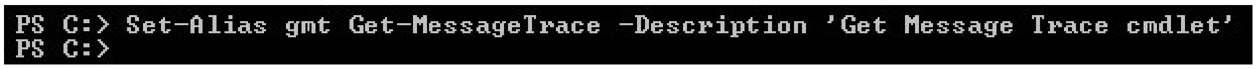
```
-------------------------- EXAMPLE 1 --------------------------

PS C:\>set-alias -name list -value get-childitem

This command creates the alias "list" for the Get-ChildItem cmdlet. After you create the alias, you can use "list"
in place of "Get-ChildItem" at the command line and in scripts.
-------------------------- EXAMPLE 2 --------------------------

PS C:\>set-alias list get-location

This command associates the alias "list" with the Get-Location cmdlet. If "list" is an alias for another cmdlet,
this command changes its association so that it now is the alias only for Get-Location.
```

Sample Usage

In practical terms, this cmdlet would likely only be used to modify existing aliases that you've created yourself. Taking some of the aliases created in the previous section, let's make sure that the aliases are ReadOnly:

```
Set-Alias dmbx Disable-Mailbox -Option ReadOnly
Set-Alias embx Enable-Mailbox -Option ReadOnly
Set-Alias gmbx Get-Mailbox -Option ReadOnly
Set-Alias nmbx New-Mailbox -Option ReadOnly
Set-Alias rmbx Remove-Mailbox -Option ReadOnly
Set-Alias smbx Set-Mailbox -Option ReadOnly
Set-Alias gcc Get-ComplianceCase -Option ReadOnly
Set-Alias ncc New-ComplianceCase -Option ReadOnly
Set-Alias rcc Remove-ComplianceCase -Option ReadOnly
Set-Alias scc Set-ComplianceCase -Option ReadOnly
```

What's interesting is that this same cmdlet ('Set-Alias') can be used to create a new alias as well. For example, if a new alias were needed for creating a new mailbox on-premises. The Set-Alias could be used to create this alias as well:

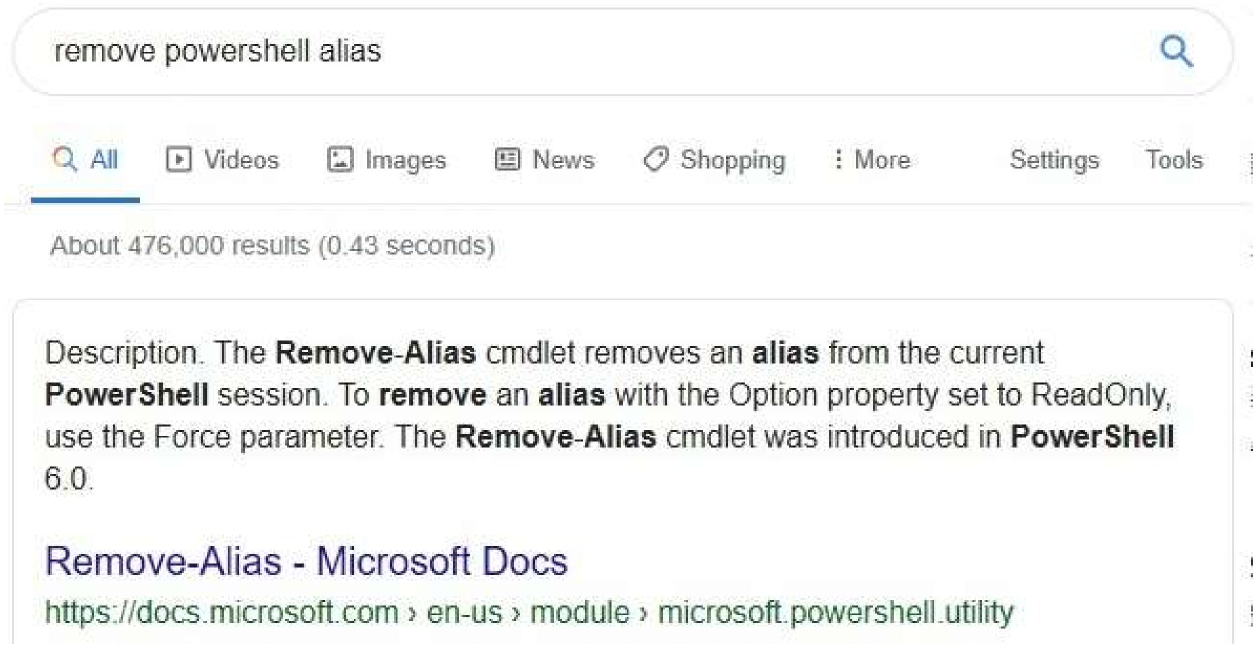


```
PS C:> Set-Alias gmt Get-MessageTrace -Description 'Get Message Trace cmdlet'
PS C:>
```

Removing an Alias

Reviewing the PowerShell cmdlets with the word 'Alias' there are no cmdlet with the word 'remove' in it. How then can an alias be removed? If the solution cannot be found in PowerShell, then searching for a solution via your favorite search engine is the next step:

```
Search string: remove powershell alias
```



About 476,000 results (0.43 seconds)

Description. The **Remove-Alias** cmdlet removes an **alias** from the current **PowerShell** session. To **remove** an **alias** with the Option property set to ReadOnly, use the Force parameter. The **Remove-Alias** cmdlet was introduced in **PowerShell** 6.0.

Remove-Alias - Microsoft Docs
https://docs.microsoft.com › en-us › module › microsoft.powershell.utility

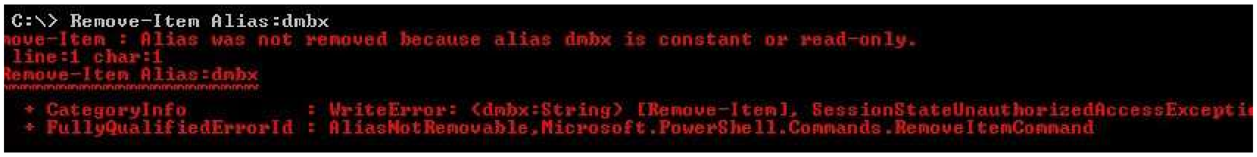
Reviewing the first link from the search, the solution to removing the alias is:

```
Remove-Item Alias:<alias to remove>
```

To remove one of the previous aliases that were created use this cmdlet:

```
Remove-Item Alias:dmbx
```

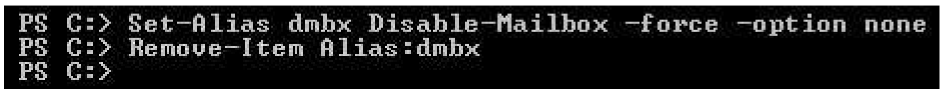
However, there is an error:



```
C:\> Remove-Item Alias:dmbx
move-Item : Alias was not removed because alias dmbx is constant or read-only.
 line:1 char:1
Remove-Item Alias:dmbx
~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : WriteError: (dmbx:String) [Remove-Item], SessionStateUnauthorizedAccessExcepti
  + FullyQualifiedErrorId : AliasNotRemovable,Microsoft.PowerShell.Commands.RemoveItemCommand
```

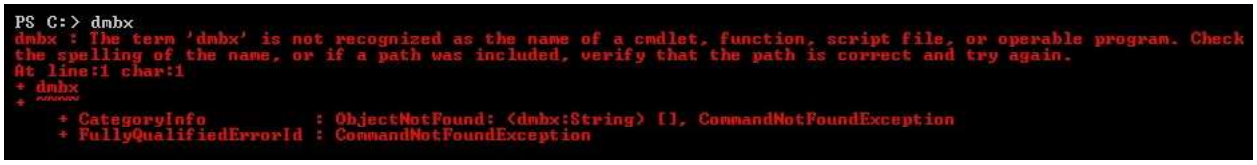
That means the ‘ReadOnly’ setting that was applied worked as expected. To remove the ReadOnly option, run this:

```
Set-Alias nrm New-RemoteMailbox -Force -Option None
Remove-Item Alias:nrm
```



```
PS C:> Set-Alias dmbx Disable-Mailbox -force -option none
PS C:> Remove-Item Alias:dmbx
PS C:>
```

Now if the alias is tried once more, PowerShell fails as the references have been removed:



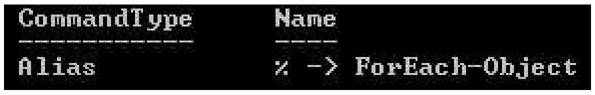
```
PS C:\> dmbx
dmbx : The term 'dmbx' is not recognized as the name of a cmdlet, function, script file, or operable program. Check
the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ dmbx
+ ~~~~
    + CategoryInfo          : ObjectNotFound: (dmbx:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

In the end, creating your own aliases is not required, nor are they necessary, but creating custom aliases may be a more efficient way to write code in PowerShell.

Foreach-Object (%)

While on the topic of PowerShell aliases, there are indeed some useful aliases that point to some rather useful cmdlets that we have not covered. One useful alias is '%'. What does the '%' symbol stand for or abbreviate in PowerShell? We can still use the Get-Alias cmdlet, but we need some criteria for finding just the '%' character in the results. If you recall from the Filtering section earlier in the book, the 'where' filter can help find the '%' symbol. From the screenshot, we also know that the field called 'Name' will contain the value:

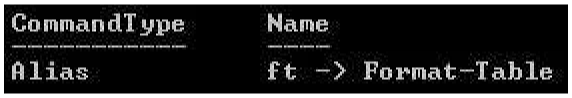
```
Get-Alias | Where {$_.Name -eq "%"}
```



```
CommandType     Name
-----------     ----
Alias           % -> ForEach-Object
```

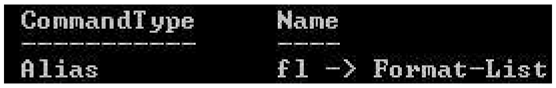
By using that cmdlet we now know that the alias % refers to Foreach-Object. Some other examples of other aliases:

```
Get-Alias | Where {$_.Name -eq "ft"}
```



```
CommandType     Name
-----------     ----
Alias           ft -> Format-Table
```

```
Get-Alias | Where {$_.Name -eq "fl"}
```



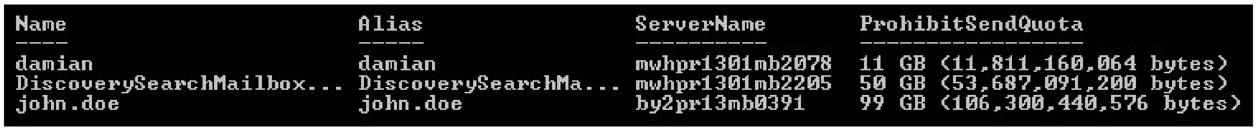
```
CommandType     Name
-----------     ----
Alias           fl -> Format-List
```

Circling back to the '%' symbol or Foreach-Object. This particular alias provides for some interesting processing of data. Take for example a scenario where we need to get the SIP address for a user's account in Exchange Online. The address exists in the EmailAddresses property for a mailbox. It is one of a number of addresses that exist there. We can write a one-liner that can pull the entire EmailAddresses property and pull out just the SIP address. In the end, a report that shows this criteria needs to be created and the PowerShell one-liner looks like this:

```
Get-Mailbox | Select-Object DisplayName, @{Expression ={$_.EmailAddresses};Label='SI\
PAddress'} | % {$Mail = $_.SIPAddress ; $Email =$Null; Foreach ($Line in $Mail) {$Ad\
dress = $Line -split ':'; $Prefix = $Address[0]; if ($Prefix -cmatch 'SIP') {$Email \
= $Address[1]}};if ($Email -eq $Null) {$Email = 'No SIP Address'};$_.SIPAddress = $E\
mail;Return $_} | FT -Auto
```

OK. Maybe that was a bit too much at once. Think of the above as what IT Management is looking for. To learn how the Foreach-Object or '%' alias fit into this, start with the results of just the 'Get-Mailbox' that we need for the replication information.

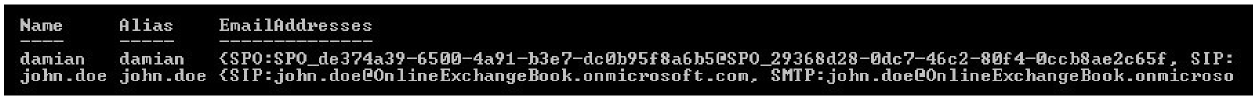
```
Get-Mailbox
```



```
Name                         Alias                 ServerName        ProhibitSendQuota
----                         -----                 ----------        -----------------
damian                       damian                mwhpr1301mb2078   11 GB (11,811,160,064 bytes)
DiscoverySearchMailbox...    DiscoverySearchMa...  mwhpr1301mb2205   50 GB (53,687,091,200 bytes)
john.doe                     john.doe              by2pr13mb0391     99 GB (106,300,440,576 bytes)
```

Notice that we get Name, Alias, ServerName and ProhibitSendQuota. This isn't what we need for our report. Yes, we can use Name and if we want an alias, but we really need to get SIP addresses. So let's user the Get-Mailbox cmdlet to reveal these mailbox properties in table format:

```
Get-Mailbox | where {$_.name -notlike 'Disc*'} | ft Name,Alias,EmailAddresses
```



```
Name     Alias    EmailAddresses
----     -----    --------------
damian   damian   {SPO:SPO_de374a39-6500-4a91-b3e7-dc0b95f8a6b5@SPO_29368d28-0dc7-46c2-80f4-0ccb8ae2c65f, SIP:
john.doe john.doe {SIP:john.doe@OnlineExchangeBook.onmicrosoft.com, SMTP:john.doe@OnlineExchangeBook.onmicroso
```

We've also filtered out the Discovery Mailbox as this is not a user mailbox.

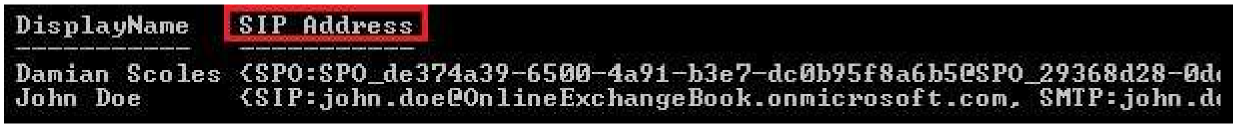
The table looks alright, however the EmailAddresses field is a mess. We also see the SIP address in the field, but we also see every other email address as well. We need to parse that field for that address. For readability sake we will also rename the field to 'SIPAddress'.

From the field data, we can determine that there are several address types stored in the EmailAddresses field - SMTP, smtp, SIP and SPO. SPO is for SharePoint Online, SMTP and smtp are our email addresses and SIP is used for Skype Online.

Let's start with the renaming of the column for the EmailAddresses property. This can be done instead of using 'Format-Table' we can use 'Select-Object' in conjunction with 'Expression' and 'Label' formatting method, like so:

```
Get-Mailbox | Where {$_.Name -NotLike 'Disc*'} | Select-Object DisplayName, @{Expres\
sion ={$_.EmailAddresses};Label='SIP Address'
```

This will then change the column heading to show as 'SIP Address' instead of 'Email Addresses':



```
DisplayName   SIP Address
-----------   -----------
Damian Scoles {SPO:SPO_de374a39-6500-4a91-b3e7-dc0b95f8a6b5@SPO_29368d28-0d
John Doe      {SIP:john.doe@OnlineExchangeBook.onmicrosoft.com, SMTP:john.d
```

Now that this is in place, we can now work with the data in the field and pull out just the SIP Address. How do we go about doing this? Well, we already have the data with we will use the 'Select-Object' cmdlet. We can now manipulate this data with a Foreach-Object, or it's alias of '%'. With this we can manipulate the data using what amounts to a PowerShell code block. In this code block we need to pull out the SIP value from the series of values in the EmailAddresses field. One of the best ways is a Foreach loop to examine each one. We only store the one value that has the key letters 'SIP' in front of it. We can also register a 'No SIP value' phrase if we do not match these letters. First is the 'Foreach-Object' to kick it off:

```
| %
```

We store all of the Email Addresses in a new variable called $Mail. Notice the EmailAddresses value can be turned in a variable by putting '$_.' in front of it. We can do this with any property if we wish to do so:

```
{$Mail = $_.EmailAddresses
```

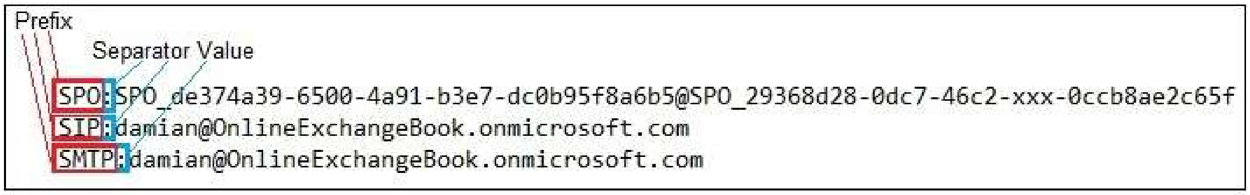
Next we configure the $SIP variable as $Null. This variable will be used to store a SIP value if found or be left empty and used to trigger the phrase about no SIP address ($Null):

```
$SIP =$Null
```

The next code section is a Foreach loop that will process each entry stored in $Email in a loop:

```
Foreach ($Line in $Mail) {
```

Each entry in the field has a Prefix (SIP, SMTP, etc) that precedes the data we need. We can split up the values in the $Line variable using a parameter called '-Split'. With this parameter we can then decide which character to separate out with. See these sample values we need to split:



```
SPO:SPO_de374a39-6500-4a91-b3e7-dc0b95f8a6b5@SPO_29368d28-0dc7-46c2-xxx-0ccb8ae2c65f
SIP:damian@OnlineExchangeBook.onmicrosoft.com
SMTP:damian@OnlineExchangeBook.onmicrosoft.com
```

We can store this separated value with the $Address like so:

```
$Address = $Line -split ':'
```

Now the $Address variable stores each part of the original field as a separate column and is numbered starting with the number '0':

```
$Prefix = $Address[0]
```

Once we store the Prefix value in the $Prefix variable we can check it to see if it contains our three special characters of 'SIP' and we also make sure that these letters are capitalized as well:

```
If ($Prefix -cmatch 'SIP') {
```

If there is a match, we can then process it by storing the address value ($Address - field 1) in a variable called SIP, like so:

```
$SIP= $Address[1]}}
```

Once this is done and all the addresses are processed, we can then check to see if the $SIP variable is empty. If it is empty, then the $SIP value is populated with 'No SIP Address' to indicate that no SIP address was found.

```
If ($SIP -eq $Null) {$SIP = 'No SIP Address'}
```

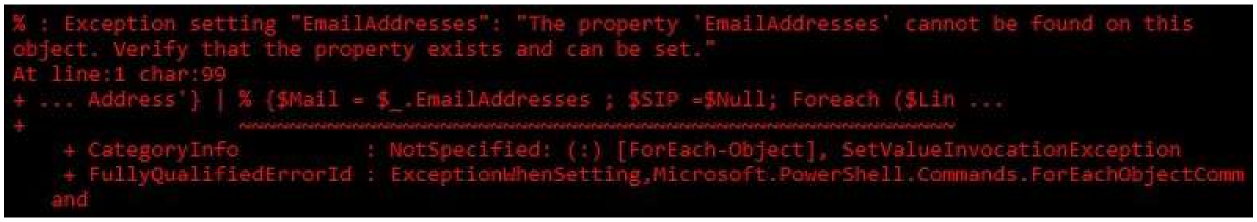
At the very end of the line, we need to then return this information back to be displayed, This requires we first use the original variable from the beginning ($EmailAddresses) as well as a 'Return' cmdlet:

```
$_.EmailAddresses = $SIP
Return $_}
```

Once all of these pieces are in place we now have a one-liner that will give us a mailbox's DisplayName and SIP address in a nicely formatted and labeled table:

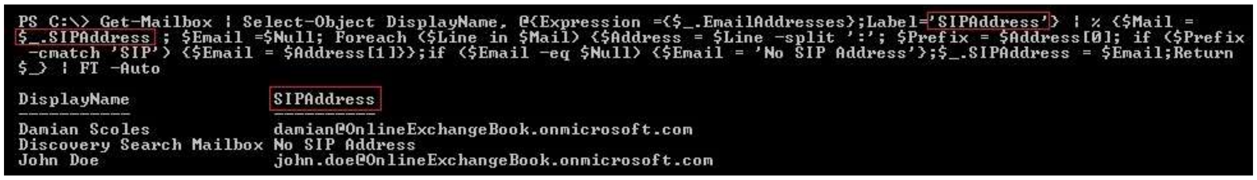
```
Get-Mailbox | Select-Object DisplayName, @{Expression ={$_.EmailAddresses};Label='SI\
P Address'} | % {$Mail = $_.EmailAddresses ; $SIP =$Null; Foreach ($Line in $Mail) {\
$Address = $Line -split ':'; $Prefix = $Address[0]; If ($Prefix -cmatch 'SIP') {$SIP\
 = $Address[1]}};If ($SIP -eq $Null) {$SIP = 'No SIP Address'};$_.EmailAddresses = $\
SIP ; Return $_} | FT -Auto
```

However, when we run this code, we get LOTS of red. What went wrong?



```
% : Exception setting "EmailAddresses": "The property 'EmailAddresses' cannot be found on this
object. Verify that the property exists and can be set."
At line:1 char:99
+ ... Address'} | % {$Mail = $_.EmailAddresses ; $SIP =$Null; Foreach ($Lin ...
+                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [ForEach-Object], SetValueInvocationException
    + FullyQualifiedErrorId : ExceptionWhenSetting,Microsoft.PowerShell.Commands.ForEachObjectComm
   and
```

Well, the error message doesn't make sense, does it? The error states that the property 'EmailAddress' cannot be found on the object. It appears that the error message is related to the Mailbox object and its EmailAddresses object. However, this is not the case. The error relates to the label of the column, which is 'SIP Address' and the variable used in the Foreach-Object which is '$EmailAddresses'. These values are different, which causes the error message to occur. Instead, these values need to match like so:

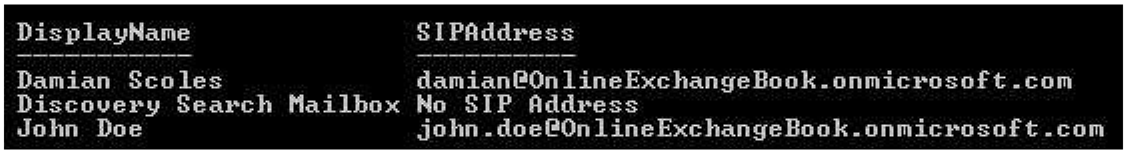


```
PS C:\> Get-Mailbox | Select-Object DisplayName, @{Expression ={$_.EmailAddresses};Label='SIPAddress'} | % {$Mail =
$_.SIPAddress ; $Email =$Null; Foreach ($Line in $Mail) {$Address = $Line -split ':'; $Prefix = $Address[0]; if ($Prefix
 -cmatch 'SIP') {$Email = $Address[1]}};if ($Email -eq $Null) {$Email = 'No SIP Address'};$_.SIPAddress = $Email;Return
$_} | FT -Auto

DisplayName             SIPAddress
-----------             ----------
Damian Scoles           damian@OnlineExchangeBook.onmicrosoft.com
Discovery Search Mailbox No SIP Address
John Doe                john.doe@OnlineExchangeBook.onmicrosoft.com
```

Code Summary

Taking all of the above information and synthesizing it, we get this long one-liner to handle the heavy lifting for us. What is nice is that we can substitute the 'SIP' phrase for 'SMTP' or even 'smtp' if we want to customize it for a different search.

```
Get-Mailbox | Select-Object DisplayName, @{Expression ={$_.EmailAddresses};Label='SI\
PAddress'} | % {$Mail = $_.SIPAddress ; $Email =$Null; Foreach ($Line in $Mail) {$Ad\
dress = $Line -split ':'; $Prefix = $Address[0]; If ($Prefix -cmatch 'SIP') {$Email \
= $Address[1]}};If ($Email -eq $Null) {$Email = 'No SIP Address'};$_.SIPAddress = $E\
mail;Return $_} | FT -Auto
```



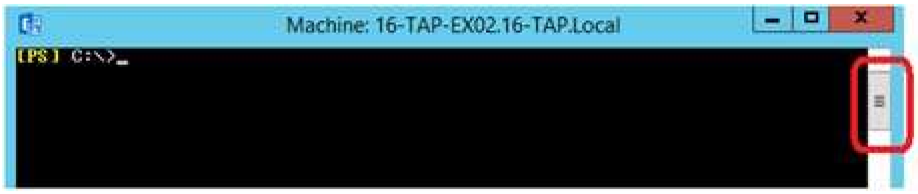
```
DisplayName             SIPAddress
-----------             ----------
Damian Scoles           damian@OnlineExchangeBook.onmicrosoft.com
Discovery Search Mailbox No SIP Address
John Doe                john.doe@OnlineExchangeBook.onmicrosoft.com
```

PowerShell Interface Customization

Working space is important in PowerShell and this means screen buffering. Why is this important? The default line buffer limit is 300 which can be too small depending on what script output of cmdlet output is being run. For example, just running 'Get-Help New-ReceiveConnector' can overrun that buffer. This makes it hard to use PowerShell to its fullest. So, just changing the buffer size will make PowerShell that much easier to work with.

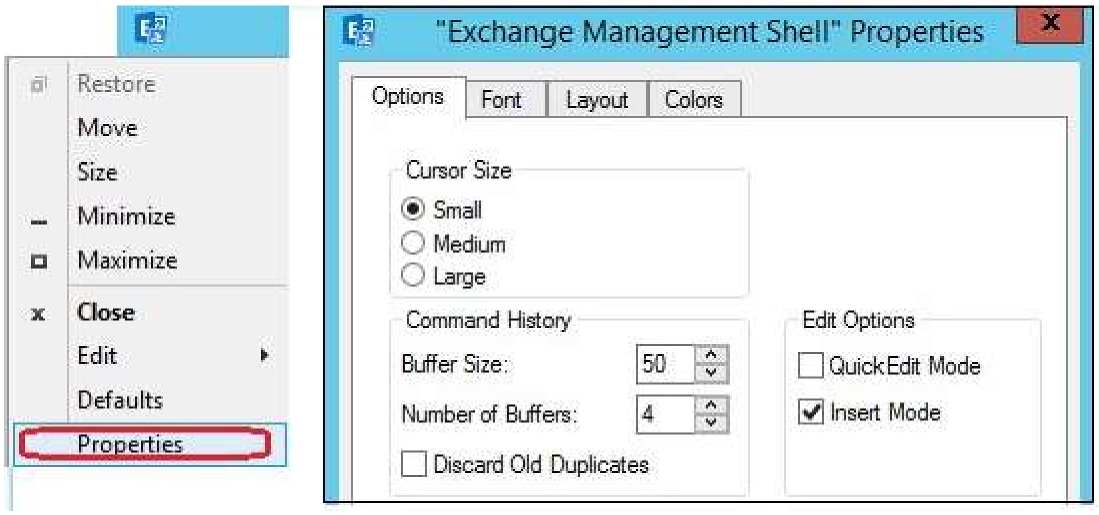
Note: These changes are local to the machine where the changes are made.

Before - 300 Character Buffer



```
[PS] C:\>_
```

To make the change, click on the icon in the upper left and select Properties (see below):

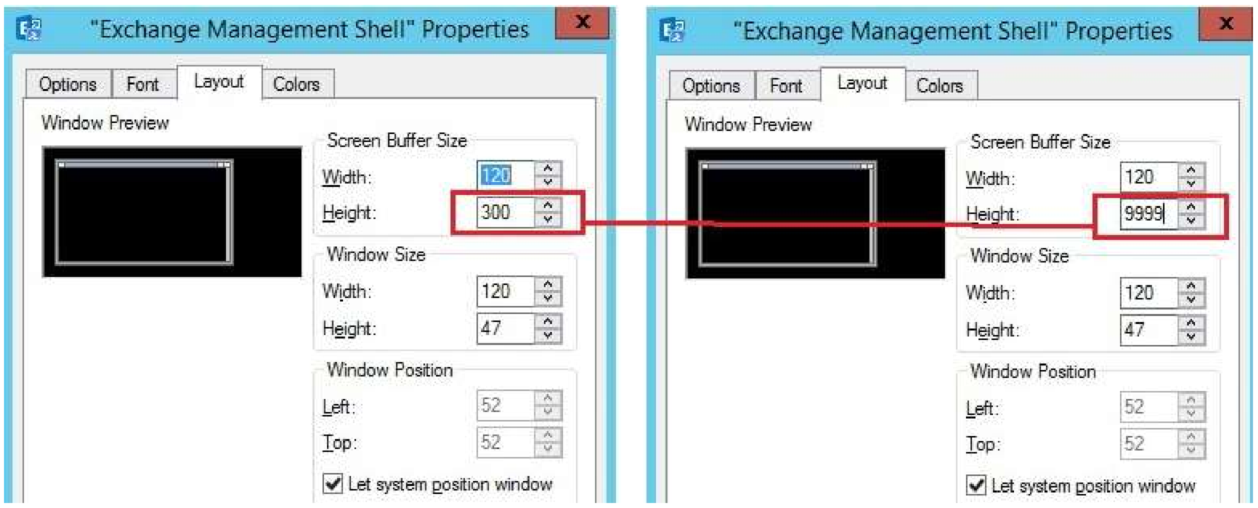


Cursor Size
- (●) Small
- ( ) Medium
- ( ) Large

Command History
- Buffer Size: 50
- Number of Buffers: 4
- [ ] Discard Old Duplicates

Edit Options
- [ ] QuickEdit Mode
- [✓] Insert Mode

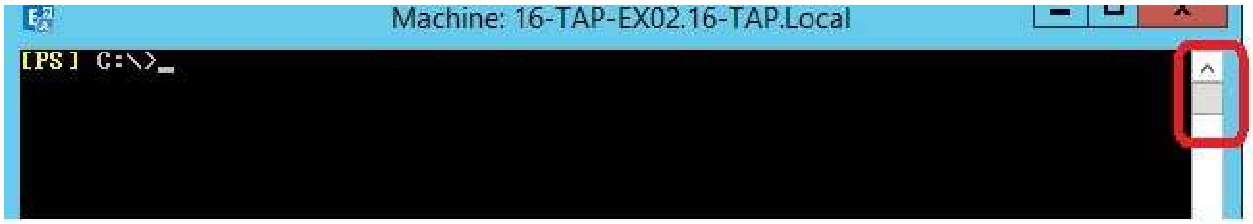
Adjust the 300 to 9999:



Screen Buffer Size
Width: 120
Height: 300
Window Size
Width: 120
Height: 47
Window Position
Left: 52
Top: 52
Let system position window

Screen Buffer Size
Width: 120
Height: 9999
Window Size
Width: 120
Height: 47
Window Position
Left: 52
Top: 52
Let system position window

After - 9999 Character Buffer



```
[PS] C:\>_
```

Notice the smaller size of the slider on the right. Now output from most, if not all cmdlets, will not exceed the window buffer size. If the output is in excess of 9999 lines, it may be better to export the results to a TXT, CSV or some other sort of file. Make sure to save these settings so we won't have to keep making this change.

In addition to the above, startup options can be created for the PowerShell window to customize it more. There are several locations for customization files for PowerShell and they vary in their functionality. The two we will work with for this chapter are:

For all users - PowerShell

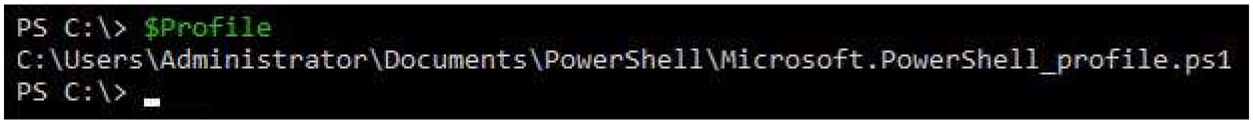
```
%windir%\system32\Windows¬PowerShell\v1.0\Microsoft.Powershell_profile.ps1
```

Current user - PowerShell

```
%UserProfile%\Documents\WindowsPowerShell\Microsoft.Powershell_profile.ps1
```

Before creating a new one, verify that one has not yet been created. Backup the old profile if needed for later. First verify the current PowerShell profile:

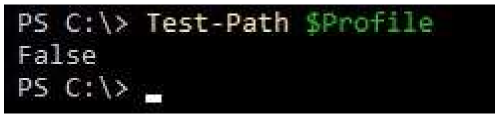
```
$Profile
```



```
PS C:\> $Profile
C:\Users\Administrator\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
PS C:\>
```

To see if the file was already created and in use (if $True, then the file exists, otherwise it does not):

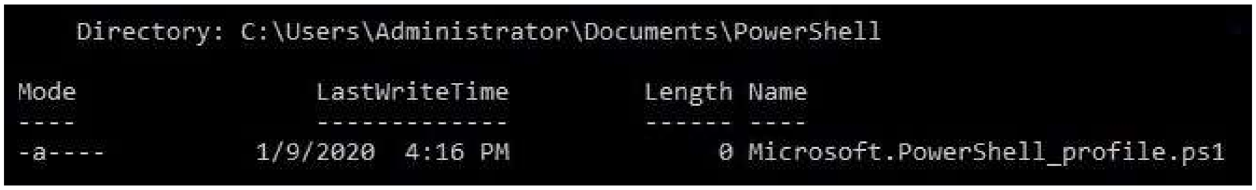
```
Test-Path $Profile
```



```
PS C:\> Test-Path $Profile
False
PS C:\>
```

In the above case, the profile has not been created and if we wish to add customizations we'll need to create our own file.

```
New-Item -Path $Profile –ItemType File –Force
```



```
    Directory: C:\Users\Administrator\Documents\PowerShell

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          1/9/2020   4:16 PM              0 Microsoft.PowerShell_profile.ps1
```

Once the file has been created you can open this in your favorite editor.

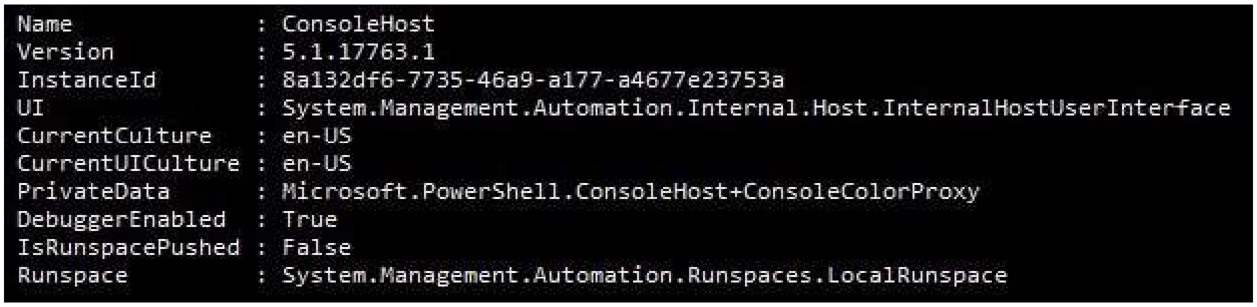
What Can Be Added to This File

The following is a list of some of the customizations that can be performed with the profile file:

- Window sizing (height and width)
- Load custom scripts
- Windows colors

Window Sizing and Coloring*
The size of the console is stored in this variable $Host which is a known variable in PowerShell.

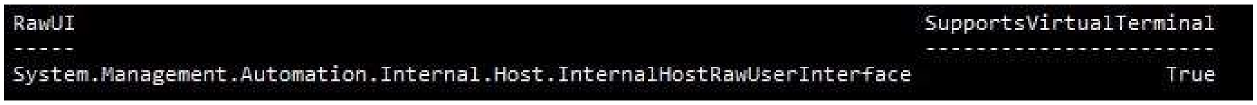
```
$Host
```



```
Name             : ConsoleHost
Version          : 5.1.17763.1
InstanceId       : 8a132df6-7735-46a9-a177-a4677e23753a
UI               : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture   : en-US
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
DebuggerEnabled  : True
IsRunspacePushed : False
Runspace         : System.Management.Automation.Runspaces.LocalRunspace
```

Notice the UI parameter is for the User Interface. To find out what is stored in it, run this:

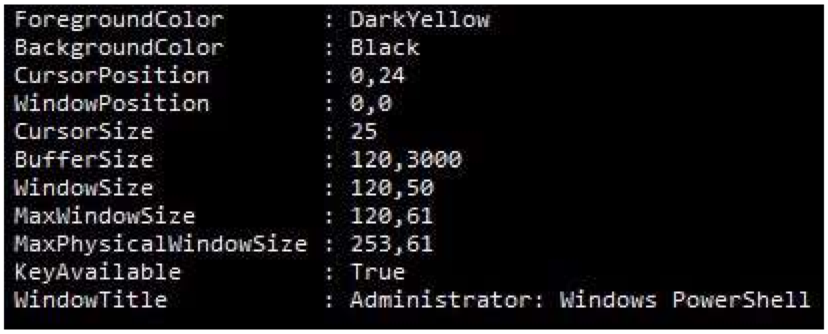
```
$Host.UI
```



```
RawUI                                                               SupportsVirtualTerminal
-----                                                               -----------------------
System.Management.Automation.Internal.Host.InternalHostRawUserInterface                True
```

That was rather unhelpful, how do we see the values stored for the UI so that changes can be made?

```
$Host.UI.RawUI
```

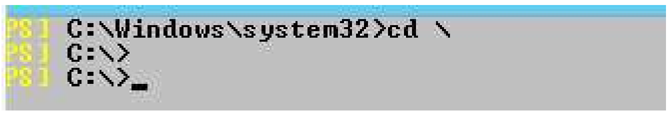


```
ForegroundColor       : DarkYellow
BackgroundColor       : Black
CursorPosition        : 0,24
WindowPosition        : 0,0
CursorSize            : 25
BufferSize            : 120,3000
WindowSize            : 120,50
MaxWindowSize         : 120,61
MaxPhysicalWindowSize : 253,61
KeyAvailable          : True
WindowTitle           : Administrator: Windows PowerShell
```

We now see the buffer size and Window Size as well as colors for the window. For this sample, the window will have a background of gray and a foreground of black. The buffer will be widened to 160 and shortened to 6000. Next the Window Size will increase to 160 and then height to 85.

```
$Shell = $Host.UI.RawUI
$Shell.ForegroundColor = "Black"
$Shell.BackgroundColor = "Gray"
$Buffer = $Shell.BufferSize
$Buffer.Width = 160
$Buffer.Height = 6000
$Shell.BufferSize = $Buffer
$Window=$Shell.WindowSize
$Window.Width = 160
$Window.Height =
$Shell.WindowSize = $Window
```

The custom colors change the PowerShell window like this:



```
PS] C:\Windows\system32>cd \
PS] C:\>
PS] C:\>_
```

In the end, when the new customized PowerShell window is opened, there may be an error message displayed. The reason is that in order to load a script with the PowerShell window, the permissions for Script Execution need to be something above Restricted, which is the default permission. For example, the 'RemoteSigned' permission will allow the script to be loaded.

```
Set-ExecutionPolicy RemoteSigned
```

That will ensure the customizations will work. Loading scripts when opening a PowerShell window requires a couple of items. First changing the location of the PowerShell window to a directory where the scripts are stored:

```
Set-Location C:\Psscripts
```

As a final step of configuring the profile script, we could run another script (below) stored the above folder:

```
.\CheckMailboxConfig.PS1
```

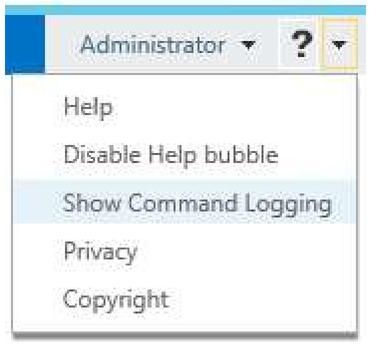
The example script above, would return various settings for our mailboxes - retention policies, quotas, OWA Mailbox Policy Mobile Device policies and more. Combining all of these steps together would results in this profile script:

```
# Load all shell parameters
$Shell = $host.UI.RawUI
$Shell.ForegroundColor = "Black"
$Shell.BackgroundColor = "Gray"
$Buffer = $Shell.BufferSize
$Buffer.Width = 160
$Buffer.Height = 6000
$Shell.BufferSize = $buffer
$Window=$Shell.WindowSize
$Window.Width = 160
$Window.Height = 50
$Shell.WindowSize = $Window
# Run Exchange Services check script
Set-Location C:\Psscripts
.\ExchangeServices.PS1
```

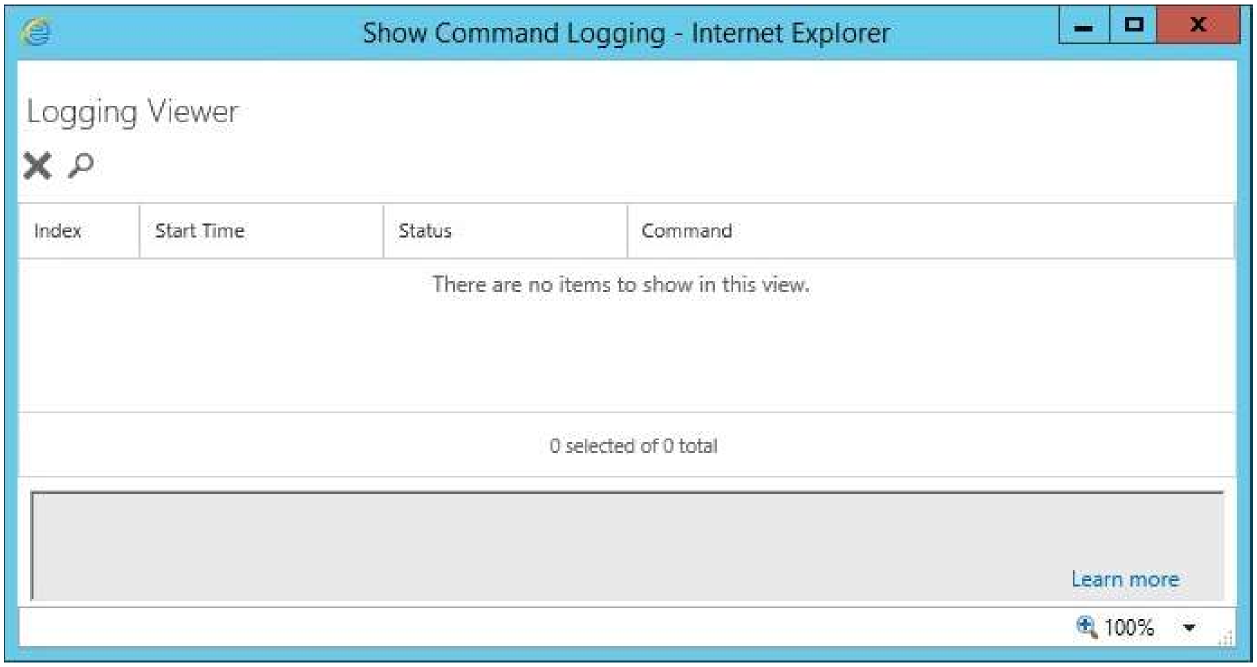
There are plenty of other options and additions that can be made to your PowerShell profile, but they will not all be listed here.

Command Logging

PowerShell is very important to Exchange Online and even the Exchange Administration Console leverages PowerShell. The question is, can this be made visible? The answer is yes, it can. What benefits does this give to those who want to use PowerShell to maintain their Exchange Online environment? For starters, when the PowerShell commands are revealed, insight may be gained into how to use the various switches and parameters when configuring an item in PowerShell. That's great, but how do we turn it on? By logging into the Exchange Administration console, click on the Administrator drop down button and select the 'Show Command Logging' option.



Once this is done, another window will pop-up so have your pop-up blocker turned off. The window looks something like this:


Show Command Logging - Internet Explorer

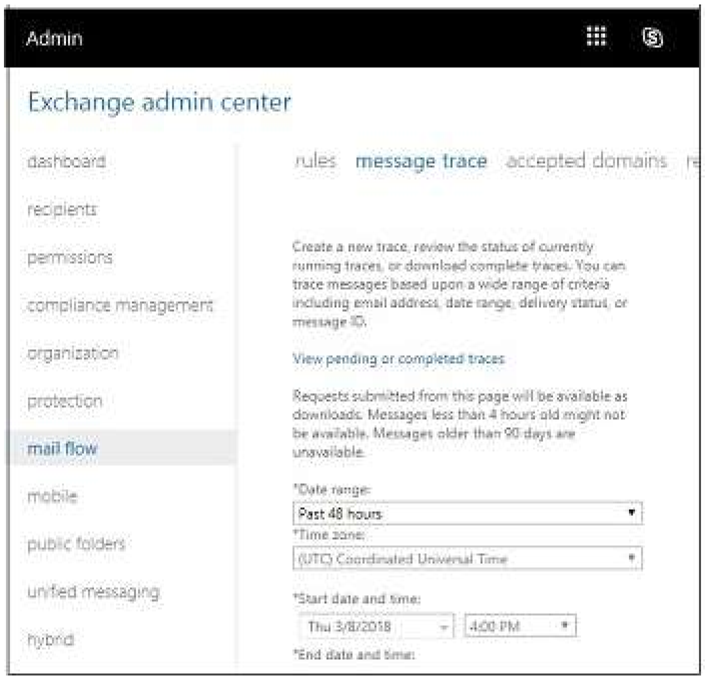
Logging Viewer

Index	Start Time	Status	Command

There are no items to show in this view.

0 selected of 0 total

Notice that all boxes in the window are completely empty. To get commands to show here, we now need to manage our Exchange Server. Let's take for an example that I want to perform a Message Trace for all messages coming into the environment. First click on 'Mail Flow' and then click on 'Message Trace':



Create a new trace, review the status of currently running traces, or download complete traces. You can trace messages based upon a wide range of criteria including email address, date range, delivery status, or message ID.

View pending or completed traces

Requests submitted from this page will be available as downloads. Messages less than 4 hours old might not be available. Messages older than 90 days are unavailable.

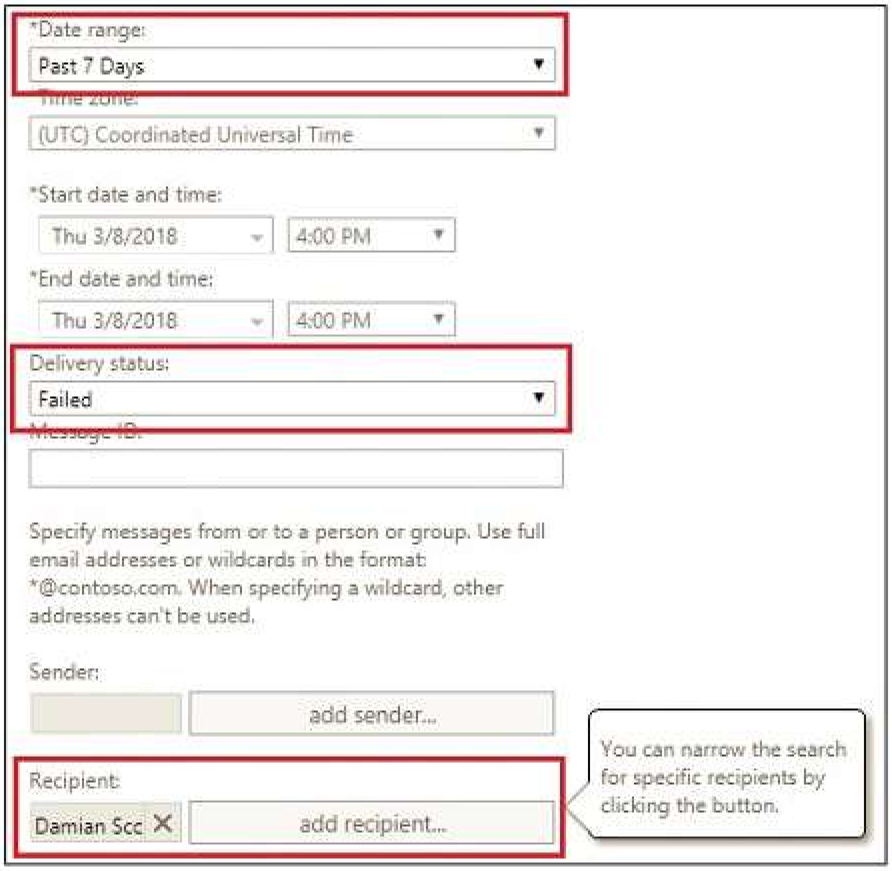
*Date range:
Past 48 hours

*Time zone:
(UTC) Coordinated Universal Time

*Start date and time:
Thu 3/8/2018 4:00 PM

*End date and time:

Now we can enter some parameters for a Message Trace - Date Range, Delivery Status and Recipient:



*Date range:
Past 7 Days

Time zone:
(UTC) Coordinated Universal Time

*Start date and time:
Thu 3/8/2018 4:00 PM

*End date and time:
Thu 3/8/2018 4:00 PM

Delivery status:
Failed

Message ID:

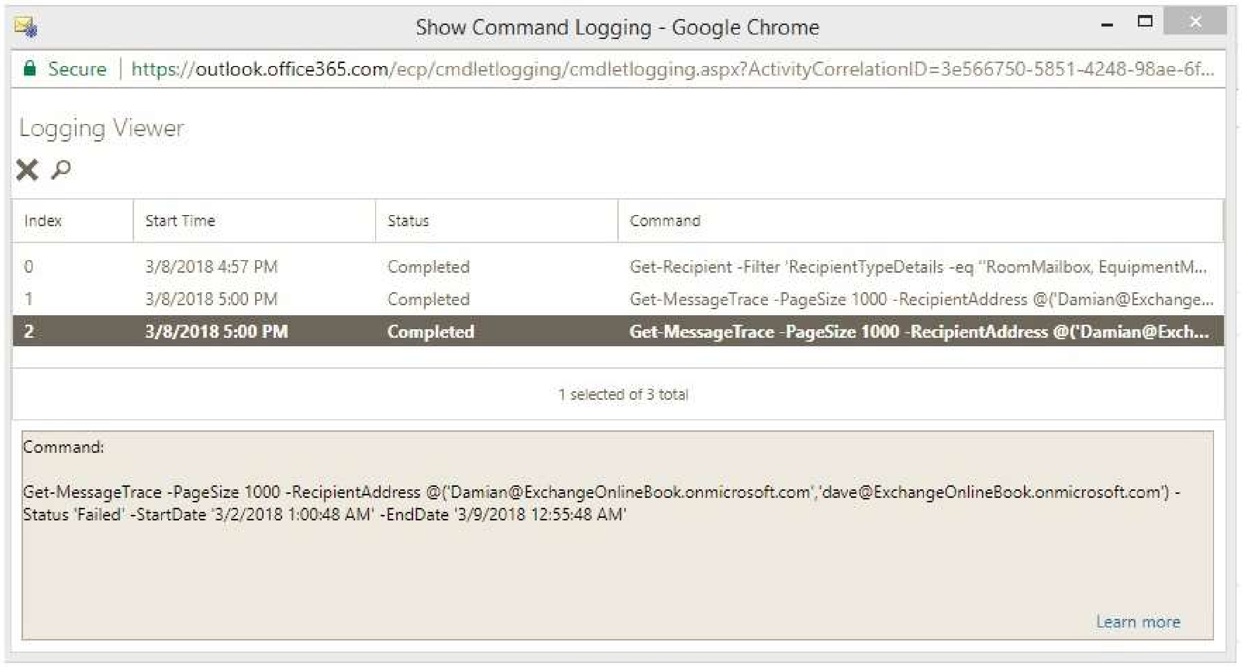
Specify messages from or to a person or group. Use full email addresses or wildcards in the format: *@contoso.com. When specifying a wildcard, other addresses can't be used.

Sender:

Recipient:
Damian Scc

You can narrow the search for specific recipients by clicking the button.

Then we can click 'Search' to get our results and then we review the Command Logging window to see what actions were taken in PowerShell:


Show Command Logging - Google Chrome

Secure | https://outlook.office365.com/ecp/cmdletlogging/cmdletlogging.aspx?ActivityCorrelationID=3e566750-5851-4248-98ae-6f...

Logging Viewer

Index	Start Time	Status	Command
0	3/8/2018 4:57 PM	Completed	Get-Recipient -Filter 'RecipientTypeDetails -eq ''RoomMailbox, EquipmentM...
1	3/8/2018 5:00 PM	Completed	Get-MessageTrace -PageSize 1000 -RecipientAddress @('Damian@Exchange...
2	3/8/2018 5:00 PM	Completed	Get-MessageTrace -PageSize 1000 -RecipientAddress @('Damian@Exch...

1 selected of 3 total

Command:

```
Get-MessageTrace -PageSize 1000 -RecipientAddress @('Damian@ExchangeOnlineBook.onmicrosoft.com','dave@ExchangeOnlineBook.onmicrosoft.com') -Status 'Failed' -StartDate '3/2/2018 1:00:48 AM' -EndDate '3/9/2018 12:55:48 AM'
```

Learn more

What can be gleaned from this information and how can an administrator use it to improve knowledge of PowerShell? First, take a look at the cmdlets that are listed. Notice that there are two cmdlets listed, Get-Recipient:

0 3/8/2018 4:57 PM Completed Get-Recipient -Filter 'RecipientTypeDetails -eq ''RoomMailbox, EquipmentM...

and Get-MessageTrace:

1 3/8/2018 5:00 PM Completed Get-MessageTrace -PageSize 1000 -RecipientAddress @('Damian@Exchange...

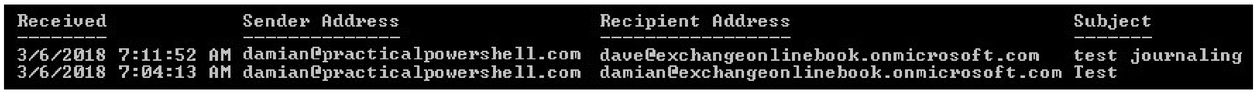
The two Get-MessageTrace cmdlets have a -PageSize of 1000 which is the default size for a Message Trace query and as we learned earlier in the book, we can manipulate how many results or records are queried. The other nice part of this Command Logging window is that we can copy and paste the cmdlets and the three that ran in our example copy and paste here as (second and third are duplicates):

```
Get-Recipient -Filter 'RecipientTypeDetails -eq ''RoomMailbox, EquipmentMailbox, Leg\
acyMailbox, LinkedMailbox, UserMailbox, MailContact, DynamicDistributionGroup, MailF\
orestContact, MailNonUniversalGroup, MailUniversalDistributionGroup, MailUniversalSe\
curityGroup, MailUser, PublicFolder, TeamMailbox, SharedMailbox, RemoteUserMailbox''\
' -Properties 'PrimarySmtpAddress,DisplayName,ArchiveGuid,AuthenticationType,Recipie\
ntType,RecipientTypeDetails,ResourceType,WindowsLiveID,Identity,ExchangeVersion,Orga\
nizationId,City,Company,CountryOrRegion,Department,Office,Title' -ResultSize 500

Get-MessageTrace -PageSize 1000 -RecipientAddress @('Damian@ExchangeOnlineBook.onmic\
rosoft.com','dave@ExchangeOnlineBook.onmicrosoft.com') -Status 'Failed' -StartDate '\
3/2/2018 1:00:48 AM' -EndDate '3/9/2018 12:55:48 AM'
```

Now we can use the above Get-MessageTrace cmdlet and try it in PowerShell to see what we get as a result. In this case the 'Failed' status provides us with zero results and is helpful in determining that these two users did not have any messages that failed to deliver. If we were to remove that condition, so that it were to look for all messages, the cmdlet would look like this:

```
Get-MessageTrace -PageSize 1000 -RecipientAddress @('Damian@ExchangeOnlineBook.onmic\
rosoft.com','dave@ExchangeOnlineBook.onmicrosoft.com') -StartDate '3/2/2018 1:00:48 \
AM' -EndDate '3/9/2018 12:55:48 AM'
```

Which does provide results:



```
Received             Sender Address                 Recipient Address                         Subject
--------             --------------                 -----------------                         -------
3/6/2018 7:11:52 AM damian@practicalpowershell.com  dave@exchangeonlinebook.onmicrosoft.com   test journaling
3/6/2018 7:04:13 AM damian@practicalpowershell.com  damian@exchangeonlinebook.onmicrosoft.com Test
```