```
#Disable-EndAccessDate.ps1
#
#Written by: Jeff Brusoe
#Last Modified: January 14, 2020
#Version: 2.0
#
#This file searches all users in the HS domain to look at their end access date (extensionAttribute1).
#For any users with an end access date that has passed, the account is disabled. See description block
#below for summary of what happens at various dates past the end access date.
#
#This file assumes a connection to Office 365 has been established. If it isn't, then it will
#look for the Connect-ToOffice365-MS3.ps1 file to attempt a connection.

#Version Updates:
#October 4, 2019
#    - Moved to GitHub directory & scheduled task
#    - Changed to new common parameters
#
#November 8, 2019
#    - Began writing data to SQL instance in Office 365
#
#January 2020
#    - Implemented automatic deletions of accounts
#    - Further testing and changes with writing to database
#    - Added primary SMTP address to DB
#    - Added original OU to DB
#    - Switched to AzureAD cmdlets

<#
.SYNOPSIS
    This file looks at all accounts that have had their passwords changed in the last 7 days. For these accounts,
    it sets the BlockCredential attribute of the MSOL User object to false.

.DESCRIPTION
    Requires
    1. Connection to the HSC tenant (Get-MsolUser etc.)
    2. Connection to Exchange online and PowerShell cmdlets (Get-Mailbox etc.)
    3. Ability to search HS domain with the Microsoft Active Directory module (Get-ADUser etc.)

    This file searches Active Directory to determine when a user is past their end access date which is stored
    in extensionAttribute1. This field is populated by SailPoint and must be changed by that system. The following
    actions are taken based on the time between the current date and the end access date.
    1. End Access Date = Current Date
        a. AD account is disabled
        b. MAPI/OWA/ActiveSync are all disabled for mailbox
        c. Set out of office reply - "The HSC account for this person is no longer active."
    2. Account Disable Date + 7 days
        a. Account is hidden in the address book
        b. AD account is removed from AD groups
        c. Add AD groups to DB
        d. Send limit to 10 kb
        e. Remove user from One Drive Members groups
        f. Set block credential to true
    3. Account Disable Date + 30 Days
        a. extensionAttribute7 is set to "No365"
        b. AD account is moved to "Deleted Accounts"
        c. Move home folder to MS OneDrive
    4. Account Disable Date + 60 Days
        a. Account is deleted.
```

```
60
61  .PARAMETER SessionTranscript
62      True = A transcript of the session (Start-Transcript) is made.
63      False = No session transcript is kept. There is really no reason to set this to
        false.
64
65  .PARAMETER PathToConnectionFile
66      This parameter is the path to the ps1 connection file to connect to the cloud.
67      The default in this file is Connect-ToOffice365-MS3.ps1 which uses the microsoft3
68      account to connect.
69
70  .PARAMETER StopOnError
71      This parameter is used for testing purposes. $true stops the program if any error
        happens.
72      For normal running, it should really be set to $false.
73
74  .PARAMETER LogFilePath
75      As the name implies, this is the path where log files are written to.
76
77  .PARAMETER DaysToKeepLogFiles
78      This is just a value to determine how long to keep log files. The default value is
        5 days.
79
80  .PARAMETER DeleteAccount
81      Specifies whether accounts that are actually deleted of just written to a log file.
82
83  .NOTES
84      Author: Jeff Brusoe
85      Last Updated by: Jeff Brusoe
86      Last Udated: January 14, 2020
87      Version: 2.0
88
89  #>
90
91  [CmdletBinding()]
92  param (
93      #Common HSC PowerShell Parameters
94      [switch]$NoSessionTranscript,
95      [string]$LogFilePath = "$PSScriptRoot\Logs",
96      [switch]$StopOnError, #$true is used for testing purposes
97      [int]$DaysToKeepLogFiles = 5, #this value used to clean old log files
98
99      #Safety Parameters
100     [switch]$DeleteAccount,
101     [int]$MaximumNewDisables = 10,
102     [int]$MaximumDeletes = 10,
103
104     #Test parameters
105     [int]$TestingUsers = 2000,
106     [int]$TestingDelay = 2, #Seconds to pause when testing program
107     [switch]$Testing,
108
109     #Database Parameters
110
        [string]$sqlPasswordPath="C:\Users\microsoft\Documents\GitHub\HSC-PowerShell-Reposito
        ry\1HSC-PowerShell-Modules\O365SqlInstance3.txt",
111     [string]$SQLServer = "hscpowershell.database.windows.net",
112     [string]$DBName = "HSCPowerShell",
113     [string]$DBUsername = "HSCPowerShell",
114     [string]$DBTableName = "DisableEndAccessDate",
115
116     #Parameters that control when various steps are performed.
117     [Alias("Step2")][int]$Step2Days = 7,
118     [Alias("Step3")][int]$Step3Days = 30,
119     [Alias("Step4")][int]$DeleteDays = 60
120 )
121
```

```powershell
#Reset environment
Clear-Host
$Error.Clear()
Set-Location $PSScriptRoot

################################
#Import HSC PowerShell Modules#
################################

#Build path to HSC PowerShell Modules
$PathToHSCPowerShellModules = $PSScriptRoot
$PathToHSCPowerShellModules =
$PathToHSCPowerShellModules.substring(0,$PathToHSCPowerShellModules.lastIndexOf("\")+1)
$PathToHSCPowerShellModules += "1HSC-PowerShell-Modules"
Write-Output $PathToHSCPowerShellModules

#Attempt to load common code module
$CommonCodeModule = $PathToHSCPowerShellModules + "\HSC-CommonCodeModule.psm1"
Write-Output "Path to common code module: $CommonCodeModule"
Import-Module $CommonCodeModule -Force -ArgumentList
$NoSessionTranscript,$LogFilePath,$true,$DaysToKeepLogFiles

#Attempt to load HSC Office 365 Module
$Office365Module = $PathToHSCPowerShellModules + "\HSC-Office365Module.psm1"
Write-Output "Path to HSC Office 365 module: $Office365Module"
Import-Module $Office365Module -Force

#Attempt to load HSC Active Directory Module
$ActiveDirectoryModule = $PathToHSCPowerShellModules + "\HSC-ActiveDirectoryModule.psm1"
Write-Output "Path to HSC Active Directory Module: $ActiveDirectoryModule"
Import-Module $ActiveDirectoryModule -Force

#Attempt to load HSC SharePoint Module
$SharePointModule = $PathToHSCPowerShellModules + "\HSC-SPOModule.psm1"
Write-Output "Path to HSC SharePoint Module: $SharePointModule"
Import-Module $SharePointModule -Force

#Attempt to load HSC SQL Module
$SQLModule = $PathToHSCPowerShellModules + "\HSC-SQLModule-Ver2.psm1"
Write-Output "Path to HSC SQL Module: $SQLModule"
Import-Module $SQLModule -Force

if ($Error.Count -gt 0)
{
    #Any errors at this point are from loading modules. Program must stop.
    Write-Warning "There was an error configuring the environment. Program is exiting."
    Exit-Commands
}

#####################################
#End of Import HSC PowerShell Modules#
#####################################

##############################
#Configure environment block#
##############################
Write-Output "Getting Parameter Information"
Get-Parameter -ParameterList $PSBoundParameters

Write-Output "Before Set-Environment"
Set-Environment
Write-Output "After Set-Environment"

Set-WindowTitle

#See this page to understand what is going on here.
#https://www.thecloudjournal.net/2016/07/create-your-own-powershell-module-for-exchange-o
```

```
nline/
ConnectTo-Office365 #from Office 365 module
Write-Output "Testing"
Import-Module ExchangeOnline -Force #comes from HSC-OFfice365Module.psm1
Write-Output "Testing2"

#Remove old log files
#Remove-OldLogFile -TXT -Path $LogFilePath -Days $DaysToKeepLogFiles -Verbose -Delete

########################################
#End of environment configuration block#
########################################

#Variable initialization
$NotFoundCount = 0
$CanBeDisabledCount = 0
$DisableCount = 0
$DoNotDisableCount = 0
$RetireeCount = 0
$ErrorCount = 0
$NewDisablesCount = 0
$DeleteCount = 0
$Count = 0 #This is the count to show where the search is.
$TotalCount = 0 #This is the total number of users found in Active Directory that will
be searched.

#########################################################
#The following try/catch blocks set constant variables.
#These should never change and are set as constants for
#safety reasons.
#########################################################
Write-Verbose "Setting file constant values."

try
{
    #This is here to eliminate a cosmetic error that would occur if the same
    #session window was used to rerun this program.
    #Setting the domain to be a constant is a safety measure too.
    Set-Variable -Name Domain -Value "hs.wvu-ad.wvu.edu" -Option Constant -Scope Global
    -ErrorAction "Stop"
    Write-Verbose "Successfully set Domain value."
}
catch [System.Management.Automation.SessionStateUnauthorizedAccessException]
{
    #This is a cosmetic error that happens when trying to set this multiple times
    during file testing.
    #It can be ignored.
    $Error.Clear()
}
catch
{
    Write-Error "Unable to set domain value. Program is ending."
    Exit-Commands
}

try
{
    #This is here to eliminate a cosmetic error that would occur if the same
    #session window was used to rerun this program.
    Set-Variable -Name DomainCN -Value "DC=hs,DC=wvu-ad,DC=wvu,DC=edu" -Option Constant
    -Scope Global -ErrorAction "Stop"
    Write-Verbose "Successfully set DomainCN"
}
catch [System.Management.Automation.SessionStateUnauthorizedAccessException]
{
    #This is a cosmetic error that happens when trying to set this multiple times
    during file testing.
```

```
248         #It can be ignored.
249         $Error.Clear()
250     }
251     catch
252     {
253         Write-Error "Unable to set domain CN value. Program is ending."
254         Exit-Commands
255     }
256
257     try
258     {
259         #This is here to eliminate a cosmetic error that would occur if the same
260         #session window was used to rerun this program.
261         Set-Variable -Name TenantName -Value "WVUHSC" -Option Constant -Scope Global
            -ErrorAction "Stop"
262         Write-Verbose "Successfully set tenant name."
263     }
264     catch [System.Management.Automation.SessionStateUnauthorizedAccessException]
265     {
266         #This is a cosmetic error that happens when trying to set this multiple times
            during file testing.
267         #It can be ignored.
268         $Error.Clear()
269     }
270     catch
271     {
272         Write-Error "Unable to set tenant name. Program is ending."
273         Exit-Commands
274     }
275
276     if ($Error.Count -gt 0)
277     {
278         Write-Warning "There was an error setting the constant values. Program is exiting."
279         Exit-Commands
280     }
281
282     Write-Output "`nConstant Values"
283     Write-Output "Domain: $Domain"
284     Write-Output "DomainCN: $DomainCN"
285     Write-Output "Tenant Name: $TenantName"
286
287     ###########################################################
288     #End of code block to set constants
289     ###########################################################
290
291     ############################
292     # Configure Log File Paths #
293     ############################
294     Write-ColorOutput -Message $("Log file directory: " + $LogFilePath) -ForegroundColor
        "Green"
295
296     $LogFilePrefix = (Get-Date -format yyyy-MM-dd-HH-mm) + "-"
297
298     #Initialize log files
299     $NotFoundFile = $LogFilePath + "\" + $LogFilePrefix + "NoEndAccessDateSet.csv" #No end
        access date
300     $CanBeDisabledFile = $LogFilePath + "\" +  $LogFilePrefix + "CanBeDisabled.csv"
301     $DoNotDisableFile = $LogFilePath + "\" + $LogFilePrefix + "DoNotDisable.csv"
302     $ErrorFile = $LogFilePath + "\" + $LogFilePrefix + "Error.txt" #This file should
        hopefully be empty after script execution.
303     $DeleteFile = $LogFilePath + "\" + $LogFilePrefix + "Delete.csv"
304     $LitigationHoldFile = $LogFilePath + "\" + $LogFilePrefix + "LitigationHold.csv"
305     $NewDisablesFile = $LogFilePath + "\" + $LogFilePrefix + "NewDisables.txt"
306     $ExcludedUserFile = $LogFilePath + "\" + $LogFilePrefix + "ExcludedUsers.csv"
307     $DirectoryMoveFIle = $LogFilePath + "\" + $LogFilePrefix + "DirectoryMove.txt"
308
309     Write-ColorOutput -foregroundcolor "Green" -Message "Log File Paths"
```

```
Write-Output "Not Found File: $NotFoundFile"
Write-Output "Can Be Disabled File: $CanBeDisabledFile"
Write-Output "Do Not Disable File: $DoNotDisableFile"
Write-Output "Error File: $ErrorFile"
Write-Output "Delete File: $DeleteFile"
Write-Output "Litigation Hold File: $LitigationHoldFile"
Write-Output "NewDisablesFile: $NewDisablesFile"
Write-Output "Directory Move File: $DirectoryMoveFile"
Write-Output "ExcludedUsersFile: $ExcludedUserFile`n`n"

#Create log files
#-force parameter causes any existing files with the same names to be overwritten
Write-ColorOutput -ForegroundColor "Green" -Message "`nCreating Log Files"

Write-Verbose "Creating Users to be Disabled File: $CanBeDisabledFile"
New-Item $CanBeDisabledFile -type file -Force

#This is for users who have a good end acces date.
Write-Verbose "Creating Do Not Disable File: $LitigationHoldFile"
New-Item $DoNotDisableFile -type file -Force

Write-Verbose "Creating EAD Not Found File: $NotFoundFile"
New-Item $NotFoundFile -type file -Force

Write-Verbose "Creating Error File: $ErrorFile"
New-Item $ErrorFile -type file -Force

Write-Verbose "Creating Delete File: $DeleteFile"
New-Item $DeleteFile -type file -Force

Write-Verbose "Creating Litigation Hold File: $LitigationHoldFile"
New-Item $LitigationHoldFile -type file -Force

Write-Verbose "Creating New Disables File: $NewDisablesFile"
New-Item $NewDisablesFile -type file -Force

Write-Verbose "Creating Excluded User File: $ExcludedUserFile"
New-Item $ExcludedUserFile -type file -Force

###############################################
# End of log file initialization code block. #
###############################################

#Generate Litigation Hold list
Write-ColorOutput -Message "`nGenerating litigation hold list" -ForegroundColor "Green"

$LitigationHold = @() #Array to hold litigation hold users
$LitigationFileArray = @()

$LHs = Get-Mailbox -ResultSize Unlimited | Where {$_.LitigationHoldEnabled -eq $true}

foreach ($LH in $LHs)
{
    Write-Output $("Litigation Hold: " + $LH.Alias)
    $LitigationHold += $LH.Alias
    $LH | select Alias,PrimarySMTPAddress,lit* | export-csv $LitigationHoldFile -Append
    -NoTypeInformation
}
#Finished generating litigation hold list and can loop through all domain users.

#Generate list of all HS users to search
Write-ColorOutput -ForegroundColor "Green" -Message "`n`nGenerating HS user list"

$PropertyArray =
"MemberOf","PasswordLastSet","lastLogonDate","msExchHideFromAddressLists","proxyAddresses
"
$PropertyArray +=
```

```
    "extensionAttribute1","extensionAttribute3","extensionAttribute7","extensionAttribute10",
    "extensionAttribute11","extensionAttribute12","extensionAttribute15"
374
375 try
376 {
377     if ($Testing)
378     {
379         $users = Get-ADUser -SearchBase $DomainCN -Properties $PropertyArray  -Filter *
        | where {(![string]::IsNullOrEmpty($_.extensionAttribute1)) -AND
        ($_.extensionAttribute10 -ne "Resource")} | select -first $TestingUsers
380         #$users = Get-ADUser pcourtney -Properties $PropertyArray
381     }
382     else
383     {
384         $users = Get-ADUser -SearchBase $DomainCN -Properties $PropertyArray  -Filter *
        | where {(![string]::IsNullOrEmpty($_.extensionAttribute1)) -AND
        ($_.extensionAttribute10 -ne "Resource")}
385     }
386
387 }
388 catch [Microsoft.ActiveDirectory.Management.ADServerDownException]
389 {
390     #This handles an exception with the following error message:
391     #"Get-ADUser : Unable to contact the server. This may be because this server does
        not exist, it is currently down, or it does not have the Active Directory Web
        Services running."
392     Write-Error "Unable to contact Active Directory server. Program is exiting."
393     Exit-Commands
394 }
395 catch [Microsoft.ActiveDirectory.Management.ADException]
396 {
397     Write-Error "Active Directory query timed out. Program is exiting."
398     Exit-Commands
399 }
400 catch
401 {
402     #More generic error
403     Write-Error "Unable to query Active Directory. Program is exiting."
404     Exit-Commands
405 }
406
407 ##Safety code used in testing
408 if ((($users | Measure).Count -ne $TestingUsers) -AND $Testing)
409 {
410     Write-Warning "Program is exiting"
411     Exit-Commands
412 }
413
414 #This is the list of user names that will be skipped in the search.
415 #It will soon be replaced with Matt's code in 5Misc-ActiveDirectoryFunctions.ps1.
416 Write-Verbose "Retrieving AD exclusion list."
417
418 #Get-SPOAExclusionList is in SharePoint misc functions file
419 #The exlusion list is a safety mesure to prevent accidentally deleting admin
420 #accounts. For example, if Trident sets all of our EAD's to today. This list is stored
    in SharePoint at this link.
421 #https://wvuhsc.sharepoint.com/PowerShellDevelopment/Lists/DoNotDisableList/AllItems.aspx
    ?viewpath=%2FPowerShellDevelopment%2FLists%2FDoNotDisableList%2FAllItems.aspx
422
423 Write-Output "Retrieving AD Exclusion List"
424
425 [int]$InitialErrorCount = $Error.Count
426 $ADExclusionList = Get-SPOADExclusionList #Stored as an array
427
428 Write-Output "`nExclusion List:"
429
430 for ($i = 0; $i -lt $ADExclusionList.Length ; $i++)
```

```
{
    $ADExclusionList[$i]
}

Write-Output "*****************************`n"

#End of exclusion list code

$TotalUsers = ($users | Measure).count
$StartTime = Get-Date

#To do: This should be moved to the common code file.
if ($StopOnError)
{
    $ErrorActionPreference = "Stop"
}

$Error.Clear() #Clears any cosmetic errors that may be from initialization before user processing

#####################################################
# Attempt connection to SQL instance in office 365 #
#####################################################
$Error.Clear()

#Decrypt SQL Password
$sqlSecureStringPassword = cat $sqlPasswordPath  | convertto-securestring
$sqlPassword = [System.Runtime.InteropServices.marshal]::PtrToStringAuto([System.Runtime.InteropServices.marshal]::SecureStringToBSTR($sqlSecureStringPassword))

#Connect to database and get everybody who has passed security compliance training.
Write-Output "Attempting Connection to SQL Server"

$SQLConn = Connect-SQL -Datasource $SQLServer -Database $DBName -Username $DBUsername -Password $sqlPassword
$SQLConnectionString = Get-ConnectionString -Datasource $SQLServer -Database $DBName -Username $DBUsername -Password $sqlPassword

Write-Output "SQL Connection Information:"
Write-Output $SQLConn

if (($Error.Count -gt 0) -OR ($SQLConn.State -eq "Closed"))
{
    #This implies an error happened connecting to SQL Server
    Write-Warning "There was an error connecting to the SQL Server. Program is exiting."
    Exit-Commands
}

##############################
# End of SQL Connection code #
##############################

#################################
# Beginning of main if/then loop #
#################################
foreach ($user in $users)
{
    Write-Output "`nDisable Count: $DisableCount"
    Write-Output "New Disables Count: $NewDisablesCount`n"

    if ($StopOnError -AND $Error.Count -gt 0)
    {
        Stop-Transcript
        Exit-Commands
    }

```

```
493         if ($Testing)
494         {
495             Start-Sleep -s $TestingDelay #Used for testing purposes
496         }
497 
498         $Count++ #This is just to indicate how far into processing the program is.
499         Write-Output "Current User count: $Count"
500         Write-Output "Total Users: $TotalUsers"
501         Write-Output $("Current User: " + $user.SamAccountName)
502 
503         Write-Progress -Activity $("Current User: " + $user.UserPrincipalName) -Status
            "Account $Count of $TotalUsers" -PercentComplete (($Count/$TotalUsers)*100)
504 
505         #Attempt to get end access date from ext1
506         try
507         {
508             Write-Output "Getting end access date from ext1"
509             [datetime]$EndAccessDate = $user.extensionAttribute1
510             Write-Output $("End Access Date: " + $EndAccessDate.toString("yyyy-MM-dd"))
511             $EndAccessDateConversionError = $false
512         }
513         catch
514         {
515             #This would happen of ext1 is $null or a string. It's a cosmetic error that
                just needs to be logged for now.
516             Write-Output "Error converting ext1 to datetime."
517             $EndAccessDateConversionError = $true
518 
519             #This is logged below
520             #Add-Content $NotFoundFile -Value $user.SamAccountName
521             #Add-Content $NotFoundFile -Value "Unable to convert ext1 to datetime"
522             #Add-Content $NotFoundFile -Value "****************************"
523 
524             $Error.Clear()
525         }
526 
527         if ($ADExclusionList -contains $user.samaccountname)
528         {
529             #Do Nothing - This is a safety measure to ensure our accounts won't be disabled
                and/or deleted.
530             Write-Output $("SamAccountName: " + $user.samaccountname)
531             Write-ColorOutput -Message "User is in excluded list... Skipping this account..."
532 
533             $user | select
                SamAccountName,Enabled,LastLogonDate,@{Name="EndAccessDate";Expression={Get-Date
                $EndAccessDate -format d}} | Export-Csv $ExcludedUserFile -Append
534         }
535         elseif ($EndAccessDateConversionError)
536         {
537             #This is the case that the End Access Date has not been set by SailPoint.
538             #For the time being, these accounts are just being logged with no other action
                taken.
539             #Some of these may be valid service accounts.
540 
541             $user | select
                SamAccountName,Enabled,LastLogonDate,@{Name="EndAccessDate";Expression={"Not
                Set"}} | Export-Csv $NotFoundFile -Append
542 
543             Write-Output $("SamAccountName: " + $user.samaccountname)
544             Write-Output $("AD Account is Enabled: " + $user.Enabled)
545 
546             if ([string]::IsNullOrEmpty($user.LastLogonDate))
547             {
548                 #user has never logged in
549                 Write-Output "Last Logon: User has never logged on"
550             }
551             else
```

```
552             {
553                 Write-Output $("Last Logon: " + $user.LastLogonDate)
554             }
555 
556             Write-Output "End Access Date is not set"
557         }
558         else
559         {
560             #This is the case where a user has a date in ext1 which must be checked.
561             Write-Output $("End Access Date: " + (Get-Date $EndAccessDate -format d))
562             Write-Output $("SamAccountName: " + $user.samaccountname)
563             Write-Output $("AD Account is Enabled: " + $user.Enabled)
564 
565             if ([string]::IsNullOrEmpty($user.LastLogonDate))
566             {
567                 #User has never logged in
568                 Write-Output "Last Logon: User has never logged on"
569             }
570             else
571             {
572                 Write-Output $("Last Logon: " + $user.LastLogonDate)
573             }
574 
575             if ($EndAccessDate -lt (Get-Date))
576             {
577                 #The end access date is before the current date.
578                 #The acocunt can begin the process of being deleted.
579 
580                 if ($NewDisablesCount -gt $MaximumNewDisables)
581                 {
582                     Write-Warning "Max new disable count has been exceeded. Program is
                    exiting."
583 
584                     Exit-Commands
585                 }
586 
587                 Write-Output $("Disabling: " + $user.SamAccountName)
588 
589                 $user | select
                    SamAccountName,Enabled,LastLogonDate,@{Name="EndAccessDate";Expression={Get-D
                    ate $EndAccessDate -format d}} | export-csv $CanBeDisabledFile -Append
590 
591                 if ($user.Enabled)
592                 {
593                     Write-Output "New user disable"
594                     $NewDisablesCount++
595                 }
596                 else
597                 {
598                     Write-Output "User is already disabled"
599                 }
600 
601                 $DisableCount++
602 
603                 #First check to see if user is already in db table
604                 $query = "select * from $DBTableName where SamAccountName = '" +
                    $user.SamAccountName + "'"
605                 Write-Output "Query: $query"
606 
607                 $SQLData = Invoke-SQLCmd -Query $query -ConnectionString $SQLConnectionString
608 
609                 #Determine Primary SMTP Address
610                 #To do: Add to AD module
611                 $ProxyAddresses = $user.proxyAddresses
612 
613                 if ($ProxyAddresses -eq $null)
614                 {
```

```
                Write-Output "No proxy addresses"
                $PrimarySMTPAddress = "None"
            }
            else
            {
                Write-Output "Proxy Addresses:"
                Write-Output $ProxyAddresses

                try
                {
                    [string]$PrimarySMTPAddress = $ProxyAddresses -cmatch "SMTP:"
                    $PrimarySMTPAddress = $PrimarySMTPAddress -replace "SMTP:",""
                    Write-Output "PrimarySMTPAddress: $PrimarySMTPAddress"

                    if ($PrimarySMTPAddress.indexOf('@') -lt 0)
                    {
                        #Verifies an actual email address is present
                        $PrimarySMTPAddress = "None"
                        Write-Output "Invalid email address found"
                    }
                    else
                    {
                        Write-Output "Valid email address found"
                    }

                    Write-Output "Primary SMTP Address: $PrimarySMTPAddress"
                }
                catch
                {
                    Write-Warning "Unable to find primary SMTP address"
                    $PrimarySMTPAddress = "None"
                }
            }

            Write-OUtput "Primary SMTP Address: $PrimarySMTPAddress"
            #End block to find primary SMTP address

            [datetime]$AccountDisableDate = Get-Date

            #Check and verify information is written to DB
            if ($SQLData -ne $null)
            {
                Write-Output "User Found"
                Write-Output $("SamAccountName From DB: " + $SQLdata.SamAccountName)
                Write-Output $("Account Disable Date from DB: " +
                $SQLData.AccountDisableDate)

                if (($SQLData.PrimarySMTPAddress -is [DBnull]) -OR
                ($SQLData.PrimarySMTPAddress.indexOf("@") -lt 0))
                {
                    #Updates previous users
                    $UpdateQuery = "UPDATE $DBTableName SET PrimarySMTPAddress =
                    '$PrimarySMTPAddress' WHERE SamAccountName = '" +
                    $user.SamAccountName + "'"

                    try
                    {
                        Write-Output "Writing primary SMTP address to DB"
                        Invoke-SQLCmd -query $UpdateQuery -ConnectionString
                        $SQLConnectionString -ErrorAction Stop
                        Write-Output "Successfully wrote primary SMTP address to DB"
                    }
                    catch
                    {
                        Write-Warning "Error writing primary SMTP address to DB"

                        if ($StopOnError)
```

```
                    {
                        Write-Warning "Program is exiting"
                        Exit-Commands
                    }
                }
            }

            if ($SQLData.AccountDisableDate -isnot [DBNull])
            {
                $AccountDisableDate = [datetime]$SQLData.AccountDisableDate
            }
        }
        else
        {
            #Need to write user to DB
            Write-Output "User not found in DB. Adding user to DB table"

            $InsertQuery = "Insert into $DBTableName
            (SamAccountName,EndAccessDate,AccountDisableDate,PrimarySMTPAddress)
            Values
            ('$($user.SamAccountName)','$($EndAccessDate.toString("yyyy-MM-dd"))','$(
            Get-Date -format yyyy-MM-dd)','$PrimarySMTPAddress')"
            Write-Output "Insert Query: $InsertQuery"

            Get-SQLQuery -Query $InsertQuery -SQLConn $SQLConn -ExecuteNonQuery

            if ($Testing)
            {
                Write-Output "Testing Delay"
                Start-Sleep -s $TestingDelay #Testing code
            }
        }

        ############################
        # Step 1a: Disable AD User #
        ############################
        Write-Output "`n`nStep 1a: Disable AD Account"

        if ($user.Enabled)
        {
            #User is enabled and will be disabled. They are being added
            #to $NewDisablesFile for logging and email alerts.

            Write-Output "Attempting to disable user"
            $user | select
            SamAccountName,LastLogonDate,@{Name="EndAccessDate";Expression={Get-Date
            $EndAccessDate -format d}} | Export-Csv $NewDisablesFile -Append

            try
            {
                $user | Disable-ADAccount -ErrorAction Stop
                Write-Output "Successfully disabled user"
            }
            catch
            {
                Write-Warning "Error disabling users"
            }

        }
        else
        {
            Write-Output "User is already disabled"
        }
        ###############
        # End Step 1a #
        ###############
```

```
738             ##############################################
739             # Step 1b: MAPI/OWA/ActiveSync set to $false #
740             ##############################################
741             Write-Output "`n`nStep 1b: Disable MAPI/OWA/ActiveSync acccess to mailbox"
742
743             #First try to find mailbox
744             $UserMailbox = $null
745             try
746             {
747                 $UserMailbox = Get-Mailbox $user.UserPrincipalName -ErrorAction Stop
748                 Write-Output "User mailbox found"
749             }
750             catch
751             {
752                 Write-Warning $("Mailbox Not Found: " + $user.UserPrincipalName)
753             }
754
755             if ($UserMailbox -eq $null)
756             {
757                 #Try using ext15@hsc.wvu.edu if UPN is not found
758                 try
759                 {
760                     [string]$ext15 = $user.extensionAttribute15
761
762                     if (![string]::IsNullOrEmpty($ext15))
763                     {
764                         $Ext15Email = $user.extensionAttribute15 + "@hsc.wvu.edu"
765                         Write-Output "Searching for: $Ext15Email"
766
767                         $UserMailbox = Get-Mailbox $Ext15Email -ErrorAction Stop
768
769                         Write-Output "Successfully found email"
770                     }
771                     else
772                     {
773                         Write-Output "extensionAttribute15 is empty"
774                     }
775                 }
776                 catch
777                 {
778                     Write-Warning "Mailbox Not Found: $Ext15Email"
779                 }
780             }
781
782             #Add search through proxy addresses here too
783             if ($UserMailbox -eq $null)
784             {
785                 #Write to DB
786             }
787             else
788             {
789                 $DisableMailbox = $false
790                 $CasMailboxAttributes =
                    "MAPIEnabled","OWAEnabled","ActiveSyncEnabled","POPEnabled","IMAPEnabled"
791
792                 $CasMailbox = $UserMailbox | Get-CasMailbox
793
794                 foreach ($CasMailboxAttribute in $CasMailboxAttributes)
795                 {
796                     Write-Output $($CasMailboxAttribute+ ": " +
                        $CasMailbox.$CasMailboxAttribute)
797
798                     if (($CasMailbox | select $CasMailboxAttribute).$CasMailboxAttribute)
799                     {
800                         $DisableMailbox = $true
801                     }
802                 }
```

```
803
804                 if ($DisableMailbox)
805                 {
806                     try
807                     {
808                         $UserMailbox | Set-CasMailbox -MAPIEnabled $false -OWAEnabled
                            $false -ActiveSyncEnabled $false -POPEnabled $false
                            -IMAPEnabled $false -ErrorAction Stop
809                         Write-Output "Successfully set CasMailbox attributes"
810                     }
811                     catch
812                     {
813                         Write-Warning "Error setting CasMailbox"
814                     }
815                 }
816                 else
817                 {
818                     Write-Output "Mailbox is already disabled"
819                 }
820             }
821             ##################
822             # End of Step 1b #
823             ##################
824
825             ####################################
826             # Step 1c: Set out of office reply #
827             ####################################
828             Write-Output "`n`nStep 1c: Set out of office reply"
829
830             try
831             {
832                 $UserMailbox | Set-MailboxAutoReplyConfiguration -AutoReplyState
                    Enabled -InternalMessage "This account has been disabled."
                    -ExternalMessage "This account has been disabled." -ErrorAction Stop
833                 Write-Output "Successfully set out of office reply"
834             }
835             catch
836             {
837                 Write-Warning "Unable to set out of office reply"
838             }
839
840             ##################
841             # End of Step 1c #
842             ##################
843
844             ########################################
845             ########################################
846             # Step 2: Account Disable Date + 7 days #
847             ########################################
848             ########################################
849
850             if ($Step2Days -gt 0)
851             {
852                 $Step2Days = -1 * $Step2Days
853             }
854
855             if ($Testing)
856             {
857                 Write-Output "Longer delay before moving to step 2"
858                 Start-Sleep -s 20
859             }
860
861             if ($AccountDisableDate -lt (Get-Date).AddDays($Step2Days))
862             {
863                 ########################################
864                 # Step 2a: Hide Mailbox in Address Book #
865                 ########################################
```

```
866
867                 Write-Output "`n`nStep 2a: Hide user from Global Address List"
868                 $HideSuccessful = $false
869
870                 if ($user.msExchHideFromAddressLists -eq $null)
871                 {
872                     Write-Output "Hidden from address lists is null"
873                 }
874                 else
875                 {
876                     Write-Output $("Hidden from address lists: " +
                    $user.msExchHideFromAddressLists)
877                 }
878
879                 if (($user.msExchHideFromAddressLists -eq $null) -OR
                    (!$user.msExchHideFromAddressLists))
880                 {
881                     try
882                     {
883                         $user | Set-ADUser -Add @{msExchHideFromAddressLists=$true}
                        -ErrorAction Stop
884                         $HideSuccessful = $true
885                     }
886                     catch
887                     {
888                         try
889                         {
890                             $user | Set-ADUser -Replace
                            @{msExchHideFromAddressLists=$true}
891                             $HideSuccessful = $true
892                         }
893                         catch
894                         {
895                             Write-Warning "Unable to hide user from address book"
896                         }
897                     }
898
899                     if ($HideSuccessful)
900                     {
901                         Write-Output "User has been hidden from address lists"
902                     }
903                     else
904                     {
905                         Write-Warning "Unable to hide user from address lists"
906                     }
907                 }
908                 else
909                 {
910                     Write-Output "User is already hidden in the address book"
911                 }
912
913                 ###################
914                 # End of Step 2a #
915                 ###################
916
917                 ###################################
918                 # Step 2b: Remove from AD groups #
919                 # Step 2c: Add groups to DB      #
920                 ###################################
921                 Write-Output "`n`nStep 2b: Remove user from AD groups"
922                 Write-Output "Step 2c: Add groups to DB (if needed)"
923
924                 Write-Output "Logging AD groups"
925                 $SelectQuery = "Select ADGroups from $DBTableName where SamAccountName
                    = '" + $user.SamAccountName + "'"
926
927                 try
```

```
928                 {
929                     $CurrentDBGroups = Invoke-SQLCmd -query $SelectQuery
                    -ConnectionString $SQLConnectionString -ErrorAction Stop
930                 }
931                 catch
932                 {
933                     Write-Warning "Error reading db groups in database. Program is
                    exiting."
934                     Exit-Commands
935                 }
936
937                 if ($CurrentDBGroups.ADGroups -isnot [DBnull])
938                 {
939
                    #https://stackoverflow.com/questions/22285149/dealing-with-system-dbn
                    ull-in-powershell
940                     Write-Output "ADGroups have already been written to DB"
941                     Write-Output "Current Groups in DB:"
942                     Write-Output $CurrentDBGroups.ADGroups
943                 }
944                 else
945                 {
946                     $UserGroups = $user.MemberOf
947                     $UpdateQuery = $null
948
949                     if ($UserGroups -eq $null)
950                     {
951                         $UpdateQuery = "UPDATE $DBTableName SET ADGroups = 'No AD
                        Groups' WHERE SamAccountName = '" + $user.SamAccountName + "'"
952                     }
953                     else
954                     {
955                         $UserGroupArray = @()
956                         foreach ($UserGroup in $UserGroups)
957                         {
958                             $TempGroupName =
                            $UserGroup.substring(0,$UserGroup.indexOf(","))
959                             $TempGroupName = $TempGroupName -replace "CN=",""
960
961                             Write-Output "Group name: $TempGroupName"
962
963                             $UserGroupArray += $TempGroupName
964                         }
965
966                         Write-Output "UserGroupArray: "
967                         Write-Output $UserGroupArray
968                         Write-Output "UserGroupArray Count: " + $UserGroupArray.Length
969
970                         $UserGroupString = $UserGroupArray -join ";"
971                         Write-Output $("User Group String Count: " +
                        $UserGroupString.Count)
972                         Write-Output "User Group String:"
973                         Write-Output $UserGroupString
974
975                         $UpdateQuery = "UPDATE $DBTableName SET ADGroups = '" +
                        ($UserGroupArray -join ";") + "' WHERE SamAccountName = '" +
                        $user.SamAccountName + "'"
976                         Write-Output `n`n"Update Query: $UpdateQuery"
977                     }
978
979                     Write-Output "Adding AD groups to DB"
980                     try
981                     {
982                         Invoke-SQLCmd -query $UpdateQuery -ConnectionString
                        $SQLConnectionString -ErrorAction Stop
983                         Write-Output "Successfully updated DB with AD groups"
984                     }
```

```
985                     catch
986                     {
987                         Write-Warning "Error writing AD groups to DB"
988                     }
989                 }
990
991                 #Now actually remove groups
992                 try
993                 {
994                     Write-Output "Removing user from AD groups"
995                     #Remove-ADPrincipalGroupMembership -Identity $User.SamAccountName
                        -MemberOf $UserGroups -Confirm:$False -ErrorAction Stop
996                     foreach ($UserGroup in $UserGroups)
997                     {
998                         Write-Output $("Removing user from group: $UserGroup")
999
1000                        try
1001                        {
1002                            Remove-ADGroupMember -Identity $UserGroup -Members
                                $user.SamAccountName -Confirm:$False -ErrorAction Stop
1003                            Write-Output "Successfully removed user from group"
1004                        }
1005                        catch
1006                        {
1007                            #This most likely is due to membership in WVU-AD groups
1008                            Write-Warning "Error removing user from this group"
1009                        }
1010                    }
1011
1012                }
1013                catch
1014                {
1015                    Write-Warning "Error removing user from groups"
1016                }
1017
1018                ########################
1019                # End of Step 2b & 2c #
1020                ########################
1021
1022                ##################################
1023                # Step 2d: Send limit to 10 kb #
1024                ##################################
1025                Write-Output "`n`n`Step 2d: Set mailbox send limit to 10 kb"
1026
1027                if ($UserMailbox -eq $null)
1028                {
1029                    Write-Output "User mailbox doesn't exist"
1030                }
1031                else
1032                {
1033                    try
1034                    {
1035                        $UserMailbox | Set-Mailbox -MaxSendSize 10kb -ErrorAction Stop
1036                        Write-Output "Successfully sent max send size"
1037                    }
1038                    catch
1039                    {
1040                        Write-Warning "Error setting max send size."
1041                    }
1042                }
1043
1044                ###################
1045                # End of Step 2d #
1046                ###################
1047
1048                ########################################################
1049                # Step 2e: Remove user from One Drive Members groups #
```

```
1050                ######################################################
1051                Write-Output "`n`nStep 2e: Remove user from OneDrive member groups
                    (OneDrive Members)"
1052                $GroupFound = $false
1053
1054                try
1055                {
1056                    Write-Output "Finding group"
1057                    $GroupObjectId = (Get-AzureADGroup -SearchString "OneDrive
                    Members").ObjectId
1058                    Write-Output "Successfully found group"
1059                    $GroupFound = $true
1060                }
1061                catch
1062                {
1063                    Write-Warning "Unable to find group"
1064
1065                }
1066
1067                if ($GroupFound)
1068                {
1069                    $UserFoundError = $false
1070
1071                    try
1072                    {
1073                        $AzureADUserObjectID = (Get-AzureADUser -SearchString
                        $user.UserPrincipalName).ObjectId
1074                        Write-Output "Azure AD User ObjectId: $AzureADUserObjectID"
1075                    }
1076                    catch
1077                    {
1078                        Write-Warning "Error finding Azure AD user"
1079                        $UserFoundError = $true
1080                    }
1081
1082                    if (!$UserFoundError)
1083                    {
1084                        try
1085                        {
1086                            Write-Output "Attempting to remove user from One Drive
                            members group"
1087                            Remove-AzureADGroupMember -ObjectId $GroupObjectId
                            -MemberId $AzureADUserObjectID -ErrorAction Stop
1088                        }
1089                        catch
1090                        {
1091                            Write-Warning "Error removing user from OneDrive Members
                            group"
1092                        }
1093                    }
1094                }
1095
1096                ##################
1097                # End of Step 2e #
1098                ##################
1099
1100                ########################################
1101                # Step 2f. Set block credential to true #
1102                ########################################
1103                Write-Output "`n`nStep 2f: Set block credential to true"
1104
1105                try
1106                {
1107                    $AzureADUser = Get-AzureADUser -SearchString
                    $user.UserPrincipalName -ErrorAction Stop
1108                    Write-Output "Successfully set block credential"
1109                }
```

```
                catch
                {
                    Write-Warning "Unable to find AzureAD Object"
                }

                if ($AzureADUser -ne $null)
                {
                    try
                    {
                        $AzureADUser | Set-AzureADUser -AccountEnabled $false
                        -ErrorAction Stop
                    }
                    catch
                    {
                        Write-Warning "Error disabling Office 365 account"
                    }
                }

                ###################
                # End of Step 2f #
                ###################

            }
            else
            {
                Write-Output "`n`nStep 2 will not be done due to account disable date."
                Write-Output "Account Disable Date: $AccountDisableDate"
            }## End of step 2

            ################################################
            ################################################
            # Begin Step 3: Account Disable Date + 30 Days #
            ################################################
            ################################################

            if ($Step3Days -gt 0)
            {
                $Step3Days = -1 * $Step3Days
            }

            if ($AccountDisableDate -lt (Get-Date).AddDays($Step3Days))
            {
                ############################################
                # Step 3a: Set extensionAttribute7 to No365 #
                ############################################
                Write-Output "`n`nStep 3a: Set extensionAttribute7 to No365 after 30
                days"

                [datetime]$AccountDisableDate = $SQLData.AccountDisableDate

                Write-Output "Account Disable Date: $AccountDisableDate"
                Write-Output $("Account Disable Date + 30 days: " +
                $AccountDisableDate.AddDays(-1*$Step3Days))

                #if ((Get-Date) -gt $AccountDisableDate.AddDays($Step3Days))
                if ($AccountDisableDate -lt (Get-Date).AddDays($Step3Days))
                {
                    Write-Output "Account disable date + 30 days has been exceeded"

                    try
                    {
                        $user | Set-ADUser -Replace @{extensionAttribute7 = "No365"}
                        -ErrorAction Stop
                        Write-Output "Successfully set extensionAttribute7 to No365"
                    }
                    catch
                    {
```

```
1173                    Write-Warning "There was an error setting extensionAttribute7"
1174                }
1175            }
1176            else
1177            {
1178                Write-Output "Account disable date + 30 days has not been exceeded
                    yet."
1179            }
1180
1181            ###############
1182            # End Step 3a #
1183            ###############
1184
1185            ########################################################
1186            # Step 3b. AD account is moved to "Deleted Accounts" #
1187            ########################################################
1188            Write-Output "`n`n`Step 3b: Move user to deleted accounts OU"
1189
1190            $CurrentDN = $user.DistinguishedName.Trim()
1191            Write-Output "Current User OU: $CurrentDN"
1192
1193            if ($CurrentDN.indexOf("OU=DeletedAccounts") -lt 0)
1194            {
1195                ### Write current OU to DB table
1196                $CurrentOU = $CurrentDN.substring($CurrentDN.indexOf(",")+1).Trim()
1197                Write-Output "Parent OU: $CurrentOU"
1198
1199                $UpdateQuery = "UPDATE $DBTableName SET OriginalOU = '$CurrentOU'
                    WHERE SamAccountName = '" + $user.SamAccountName + "'"
1200                Write-Output `n`n"Update Query: $UpdateQuery"
1201
1202                try
1203                {
1204                    Invoke-SQLCmd -query $UpdateQuery -ConnectionString
                        $SQLConnectionString -ErrorAction Stop
1205                    Write-Output "Successfully updated DB with user's current OU"
1206                }
1207                catch
1208                {
1209                    Write-Warning "Error writing users's current OU to DB"
1210
1211                    if ($StopOnError)
1212                    {
1213                        Exit-Commands
1214                    }
1215                }
1216            }
1217            else
1218            {
1219                Write-Output "User has already been moved to the deleted accounts OU"
1220            }
1221            ### Current OU has now been written to DB
1222
1223            [string]$DeletedAccountsOU =
                "OU=DeletedAccounts,DC=hs,DC=wvu-ad,DC=wvu,DC=edu"
1224            Write-Output "Deleted Accounts OU: $DeletedAccountsOU"
1225
1226            if ($user.DistinguishedName.indexOf("OU=DeletedAccounts") -lt 0)
1227            {
1228                try
1229                {
1230                    $user | Move-ADObject -TargetPath $DeletedAccountsOU
                        -ErrorAction Stop
1231                    Write-Output "Successfully moved AD user to $DeletedAccountsOU"
1232                }
1233                catch
1234                {
```

```
                    Write-Warning "There was an error moving user to
                    $DeletedAccountsOU"

                    if ($StopOnError)
                    {
                        Exit-Commands
                    }
                }
            }
            else
            {
                Write-Output "User has already been moved to deleted accounts OU."
            }

            ###############
            # End Step 3b #
            ###############

            ###############################################
            # Step 3c. Move home folder to MS OneDrive #
            ###############################################

            #Testing is still going on for this step. Need to figure out how to
            uniquely determine
            #user directory as well as how to upload to OneDrive.

            Write-Output "`n`nStep 3c: Move home folder to MS OneDrive"

            Add-Content -Path $DirectoryMoveFile -Value $user.SamAccountName

            try
            {
                Write-Output "Attempting to move user directory"
                #Need to figure this part out
                Write-Output "Home directory has been moved"
            }
            catch
            {
                Write-Warning "Error moving user directory"
            }

            ###############
            # End Step 3c #
            ###############
        }
        else
        {
            Write-Output "`n`n`Step 3 will not be run at this time due to the
            account disable date"
            Write-Output "Account Disable Date: $AccountDisableDate"

        }

        ################################################
        ################################################
        # Begin Step 4: Account Disable Date + 60 Days #
        ################################################
        ################################################

        Write-Output "`n`nStep 4: Deleting AD Account"

        if ($DeleteDays -gt 0)
        {
            $DeleteDays = -1*$DeleteDays
        }

        if ($AccountDisableDate -lt (Get-Date).AddDays($DeleteDays))
```

```
1299            {
1300                ############################
1301                # Step 4: Delete AD account #
1302                ############################
1303
1304                $SamAccountName = $user.SamAccountName
1305
1306                Write-Output "Deleting: $SamAccountName"
1307                Add-Content -Path $DeleteFile -Value $SamAccountName
1308
1309                try
1310                {
1311                    #Add account delete date to DB
1312                    $AccountDeleteDate = (Get-Date -format yyyy-MM-dd).toString()
1313                    Write-Output "Account Delete Date: $AccountDeleteDate"
1314
1315                    $UpdateQuery = "UPDATE $DBTableName SET AccountDeleteDate =
                        '$AccountDeleteDate' WHERE SamAccountName = '" +
                        $user.SamAccountName + "'"
1316                    Write-Output `n`n"Update Query: $UpdateQuery"
1317
1318                    Write-Output "Setting account delete date in DB"
1319                    Invoke-SQLCmd -query $UpdateQuery -ConnectionString
                        $SQLConnectionString -ErrorAction Stop
1320                    Write-Output "Successfully updated account delete date"
1321                }
1322                catch
1323                {
1324                    Write-Warning "Error writing users's account delete date"
1325
1326                    if ($StopOnError)
1327                    {
1328                        Exit-Commands
1329                    }
1330                }
1331
1332                try
1333                {
1334                    $DeleteCount++
1335                    Write-Output "Delete Count: $DeleteCount"
1336
1337                    if ($DeleteCount -lt $MaximumDeletes)
1338                    {
1339                        if (!Testing)
1340                        {
1341                            Write-Output "Attempting to delete AD user account"
1342                            #$user | Remove-ADUser -ErrorAction Stop -Confirm:$false
1343                            Write-Output "AD user account successfully deleted."
1344                        }
1345                    }
1346                    else
1347                    {
1348                        Write-Output "Maximum deletes have been reached."
1349                        Write-Output "User will not be deleted."
1350                    }
1351
1352                    Write-Output "Successfully deleted AD account"
1353                }
1354                catch
1355                {
1356                    Write-Warning "Error deleting AD account"
1357
1358                    if ($StopOnError)
1359                    {
1360                        Exit-Commands
1361                    }
1362                }
```

```
1363                }
1364                else
1365                {
1366                    Write-Output "Account Disable Date: $AccountDisableDate"
1367                    Write-Output "AD account will not be deleted at this time."
1368                }
1369
1370            }
1371        }
1372
1373        Write-Output "*******************************"
1374
1375    } #End of main for loop
1376
1377    $EndTime = Get-Date
1378    $TotalTime = $EndTime - $StartTime
1379
1380    Write-ColorOutput -ForegroundColor "Green" -Message "`nSummary Output"
1381    "Processing took: " + $TotalTime.ToString("hh\:mm\:ss")
1382    "Disable Count: " + $DisableCount.toString()
1383    "New Disable Count: $NewDisablesCount"
1384    "Do Not Disable Count: " + $DoNotDisableCount.toString()
1385
1386    if (!$NoSessionTranscript)
1387    {
1388        Write-Verbose "Stopping session transcript"
1389
1390        Stop-Transcript
1391    }
1392
1393    Exit
```