Changing Ownership of File or Folder Using PowerShell

https://learn-powershell.net/2014/06/24/changing-ownership-of-file-or-folder-using-powershell/

While working on a project recently, I needed to find an easy way to take ownership of a profile folder and its subfolders to allow our support staff to either delete the profile or be able to traverse the folder to help troubleshoot issues. Typically, one could use Explorer to find the folder and then take ownership and be done with it. But the goal was to come up with a command line solution that not only worked quickly, but didn't miss out on a file or folder.

The brief background on this is that roaming profiles sometimes would become inaccessible to our support staff in that only the user account and System would have access to the profile folder and its sub-folders and files. Also, ownership of those objects were by the user account. This created issues with deleting accounts and troubleshooting profile related issues.

Before showing the solution that I came up with, I will run down a list of attempts which never quite met my requirements and why.

Using Takeown.exe

This was actually my initial idea as I allows for recursive actions and lets me specify to grant ownership to Builtin\Administrators. Sure it wasn't a PowerShell approach, but it met the requirements of what I wanted to do…or so I thought.

The first problem is that it is slow. I kicked it off on my own profile (because it is always more fun to test on yourself than others) and found that it would take upwards of 10 minutes vs. the ~2 minute UI approach. Obviously this is an issue if I expect to have this used as part of my project for others to take ownership on profiles which would more than likely have more items than my profile. I still decided to press forward with this and later found the second issue: takeown.exe would not reliably grant ownership completely down the tree of subfolders. This was a huge issue and would not be acceptable with the customer.

Take Ownership using PowerShell and Set-ACL

The next idea was to grab the ACL object of a folder elsewhere in the user's home directory that had good permissions and then change the owner in that ACL object to 'Builtin\Administrators" and the apply it to the profile folder.

```
$ACL = Get-ACL .\smithb
$Group = New-Object System.Security.Principal.NTAccount("Builtin", "Administrators")
$ACL.SetOwner($Group)
Set-Acl -Path .\smithb\profile.v2 -AclObject $ACL
```

Sounds good, right? Well, not really due to some un-foreseen issues. Because the accounts do not have the proper user rights (**seTakeOwnershipPrivilege**, **SeRestorePrivilege** and **SeBackupPrivilege**), this would fail right away with an 'Access Denied' error. Fine, I can add those privileges if needed and continue on from there. Well, it doesn't quite work that way either because only the directories would propagate these permissions but the files wouldn't get ownership.

Set-Owner Function

The final thing that I came up with followed a similar idea as my second attempt, but makes sure to allow for recursion and files and folders as well as allowing either 'Builting\Administrators' or another account to have ownership of files and folders. To do this I dove into the Win32 API to first allow the account to elevate the tokens that I have mentioned before.

```
Try {
[void][TokenAdjuster]
} Catch {
$AdjustTokenPrivileges = @"
using System;
```

```
using System.Runtime.InteropServices;

public class TokenAdjuster
{
    [DllImport("advapi32.dll", ExactSpelling = true, SetLastError = true)]
    internal static extern bool AdjustTokenPrivileges(IntPtr htok, bool disall,
    ref TokPriv1Luid newst, int len, IntPtr prev, IntPtr relen);
    [DllImport("kernel32.dll", ExactSpelling = true)]
    internal static extern IntPtr GetCurrentProcess();
    [DllImport("advapi32.dll", ExactSpelling = true, SetLastError = true)]
    internal static extern bool OpenProcessToken(IntPtr h, int acc, ref IntPtr
    phtok);
    [DllImport("advapi32.dll", SetLastError = true)]
    internal static extern bool LookupPrivilegeValue(string host, string name,
    ref long pluid);
    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    internal struct TokPriv1Luid
    {
        public int Count;
        public long Luid;
        public int Attr;
    }
    internal const int SE_PRIVILEGE_DISABLED = 0x00000000;
    internal const int SE_PRIVILEGE_ENABLED = 0x00000002;
    internal const int TOKEN_QUERY = 0x00000008;
    internal const int TOKEN_ADJUST_PRIVILEGES = 0x00000020;
    public static bool AddPrivilege(string privilege)
    {
        try
        {
            bool retVal;
            TokPriv1Luid tp;
            IntPtr hproc = GetCurrentProcess();
            IntPtr htok = IntPtr.Zero;
            retVal = OpenProcessToken(hproc, TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, ref htok);
            tp.Count = 1;
            tp.Luid = 0;
            tp.Attr = SE_PRIVILEGE_ENABLED;
            retVal = LookupPrivilegeValue(null, privilege, ref tp.Luid);
            retVal = AdjustTokenPrivileges(htok, false, ref tp, 0, IntPtr.Zero, IntPtr.Zero);
            return retVal;
        }
        catch (Exception ex)
        {
            throw ex;
        }
    }
    public static bool RemovePrivilege(string privilege)
        {
        try
        {
            bool retVal;
            TokPriv1Luid tp;
            IntPtr hproc = GetCurrentProcess();
            IntPtr htok = IntPtr.Zero;
            retVal = OpenProcessToken(hproc, TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, ref htok);
            tp.Count = 1;
            tp.Luid = 0;
            tp.Attr = SE_PRIVILEGE_DISABLED;
            retVal = LookupPrivilegeValue(null, privilege, ref tp.Luid);
            retVal = AdjustTokenPrivileges(htok, false, ref tp, 0, IntPtr.Zero, IntPtr.Zero);
            return retVal;
        }
        catch (Exception ex)
        {
            throw ex;
        }
    }
}
"@
Add-Type $AdjustTokenPrivileges
}

#Activate necessary admin privileges to make changes without NTFS perms
[void][TokenAdjuster]::AddPrivilege("SeRestorePrivilege") #Necessary to set Owner Permissions
[void][TokenAdjuster]::AddPrivilege("SeBackupPrivilege") #Necessary to bypass Traverse Checking
```

```
[void][TokenAdjuster]::AddPrivilege("SeTakeOwnershipPrivilege") #Necessary to override
FilePermissions
```

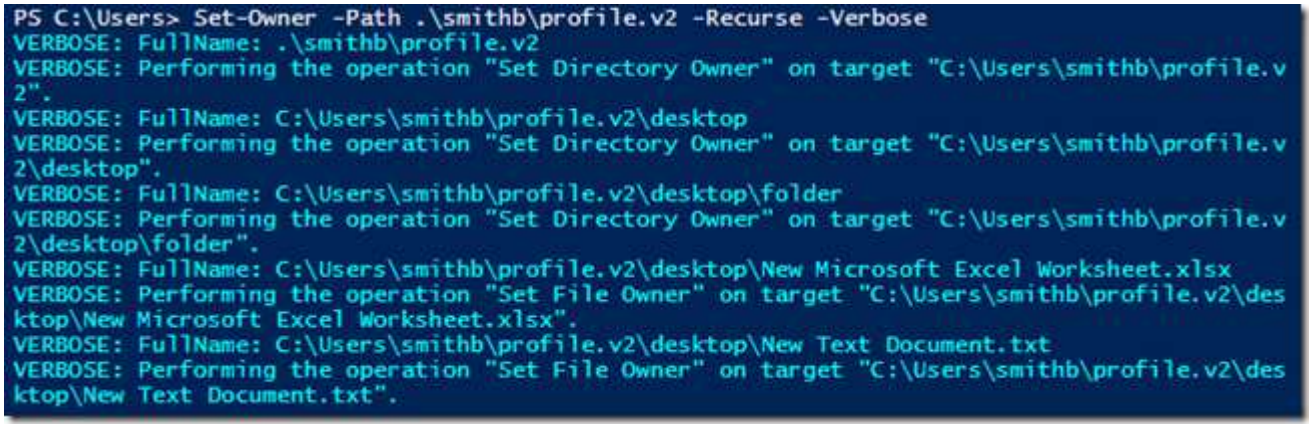
This allows me to traverse the directory tree and set ownership on the files and folders. If I cannot take ownership on a file or folder (because inheritance is not allowed from the parent folder), then it moves up a level to grant Full Control to to parent folder, thus allowing me to take ownership on the folder or file below it.

```
Process {
    ForEach ($Item in $Path) {
        Write-Verbose "FullName: $Item"
        #The ACL objects do not like being used more than once, so re-create them on the Process
block
        $DirOwner = New-Object System.Security.AccessControl.DirectorySecurity
        $DirOwner.SetOwner([System.Security.Principal.NTAccount]$Account)
        $FileOwner = New-Object System.Security.AccessControl.FileSecurity
        $FileOwner.SetOwner([System.Security.Principal.NTAccount]$Account)
        $DirAdminAcl = New-Object System.Security.AccessControl.DirectorySecurity
        $FileAdminAcl = New-Object System.Security.AccessControl.DirectorySecurity
        $AdminACL = New-Object
System.Security.AccessControl.FileSystemAccessRule('Builtin\Administrators','FullControl','Containe
rInherit,ObjectInherit','InheritOnly','Allow')
        $FileAdminAcl.AddAccessRule($AdminACL)
        $DirAdminAcl.AddAccessRule($AdminACL)
        Try {
            $Item = Get-Item -LiteralPath $Item -Force -ErrorAction Stop
            If (-NOT $Item.PSIsContainer) {
                If ($PSCmdlet.ShouldProcess($Item, 'Set File Owner')) {
                    Try {
                        $Item.SetAccessControl($FileOwner)
                    } Catch {
                        Write-Warning "Couldn't take ownership of $($Item.FullName)! Taking
FullControl of $($Item.Directory.FullName)"
                        $Item.Directory.SetAccessControl($FileAdminAcl)
                        $Item.SetAccessControl($FileOwner)
                    }
                }
            } Else {
                If ($PSCmdlet.ShouldProcess($Item, 'Set Directory Owner')) {
                    Try {
                        $Item.SetAccessControl($DirOwner)
                    } Catch {
                        Write-Warning "Couldn't take ownership of $($Item.FullName)! Taking
FullControl of $($Item.Parent.FullName)"
                        $Item.Parent.SetAccessControl($DirAdminAcl)
                        $Item.SetAccessControl($DirOwner)
                    }
                }
                If ($Recurse) {
                    [void]$PSBoundParameters.Remove('FullName')
                    Get-ChildItem $Item -Force | Set-Owner @PSBoundParameters
                }
            }
        } Catch {
            Write-Warning "$($Item): $($_.Exception.Message)"
        }
    }
}
End {
    #Remove priviledges that had been granted
    [void][TokenAdjuster]::RemovePrivilege("SeRestorePrivilege")
    [void][TokenAdjuster]::RemovePrivilege("SeBackupPrivilege")
    [void][TokenAdjuster]::RemovePrivilege("SeTakeOwnershipPrivilege")
}
```

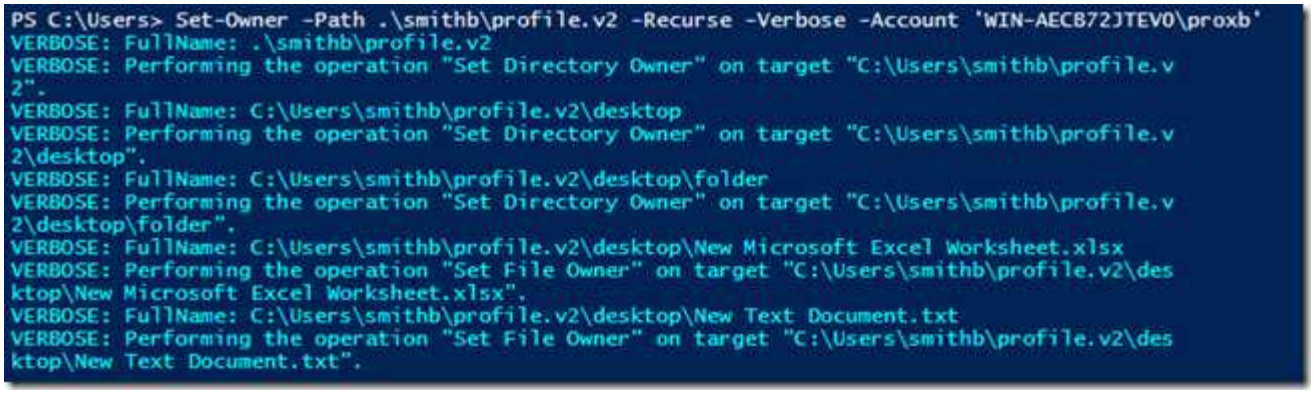
Using this approach, I was able to accurately take ownership on all of the items as well as not facing major slowdown (it was roughly 30 seconds slower than the UI approach). Seemed like a good tradeoff to me.

Here are a couple of examples of the function in action:

```
Set-Owner -Path .\smithb\profile.v2 -Recurse -Verbose
```




```
Set-Owner -Path .\smithb\profile.v2 -Recurse -Verbose -Account 'WIN-AECB72JTEV0\proxb'
```



The function is available to download from the following link:

http://gallery.technet.microsoft.com/scriptcenter/Set-Owner-ff4db177

Set-Owner

This function allows you to change the owner of a file(s) or folder(s) to another user or group. This is similiar to the takeown.exe command that is available from the command prompt, but in PowerShell. The function will elevate the token of the account (only in console session

This function allows you to change the owner of a file(s) or folder(s) to another user or group. This is similiar to the takeown.exe command that is available from the command prompt, but in PowerShell. The function will elevate the token of the account (only in console session) running this command to allow access to change the ownership of the file or folder. The default value of -Account is *'Builtin\Administrators'* but can be changed to any allowed user or group. You can also recursively go through files and folders and apply the new ownership as well.

Related blog post: http://learn-powershell.net/2014/06/24/changing-ownership-of-file-or-folder-using-powershell/

Remember to dot source the script to load the function into the console session.

PowerShell
Edit|Remove
powershell

```
. .\Set-Owner.ps1
. .\Set-Owner.ps1
```

Change ownership of folder and subfolders/files.

PowerShell

```
Set-Owner -Path C:\temp -Recurse
```

Allow for pipeline input and set owner to another account.

PowerShell

```
Get-ChildItem C:\Temp |
Set-Owner -Recurse -Account 'Domain\bprox'
```

Verified on the following platforms

Windows 10	No
Windows Server 2012	Yes
Windows Server 2012 R2	No
Windows Server 2008 R2	Yes
Windows Server 2008	Yes
Windows Server 2003	Yes
Windows 8	Yes
Windows 7	Yes
Windows Vista	Yes
Windows XP	No
Windows 2000	No

This script is tested on these platforms by the author. It is likely to work on other platforms as well. If you try it and find that it works on another platform, please add a note to the script discussion to let others know.

Online peer support For online peer support, join The Official Scripting Guys Forum! To provide feedback or report bugs in sample scripts, please start a new discussion on the Discussions tab for this script.


```
Function Set-Owner {
    <#
        .SYNOPSIS
            Changes owner of a file or folder to another user or group.

        .DESCRIPTION
            Changes owner of a file or folder to another user or group.

        .PARAMETER Path
            The folder or file that will have the owner changed.

        .PARAMETER Account
            Optional parameter to change owner of a file or folder to specified account.

            Default value is 'Builtin\Administrators'

        .PARAMETER Recurse
            Recursively set ownership on subfolders and files beneath given folder.

        .NOTES
            Name: Set-Owner
            Author: Boe Prox
            Version History:
                 1.0 - Boe Prox
```

```
                - Initial Version

    .EXAMPLE
        Set-Owner -Path C:\temp\test.txt

        Description
        -----------
        Changes the owner of test.txt to Builtin\Administrators

    .EXAMPLE
        Set-Owner -Path C:\temp\test.txt -Account 'Domain\bprox

        Description
        -----------
        Changes the owner of test.txt to Domain\bprox

    .EXAMPLE
        Set-Owner -Path C:\temp -Recurse

        Description
        -----------
        Changes the owner of all files and folders under C:\Temp to Builtin\Administrators

    .EXAMPLE
        Get-ChildItem C:\Temp | Set-Owner -Recurse -Account 'Domain\bprox'

        Description
        -----------
        Changes the owner of all files and folders under C:\Temp to Domain\bprox
#>
[cmdletbinding(
    SupportsShouldProcess = $True
)]
Param (
    [parameter(ValueFromPipeline=$True,ValueFromPipelineByPropertyName=$True)]
    [Alias('FullName')]
    [string[]]$Path,
    [parameter()]
    [string]$Account = 'Builtin\Administrators',
    [parameter()]
    [switch]$Recurse
)
Begin {
    #Prevent Confirmation on each Write-Debug command when using -Debug
    If ($PSBoundParameters['Debug']) {
        $DebugPreference = 'Continue'
    }
    Try {
        [void][TokenAdjuster]
    } Catch {
        $AdjustTokenPrivileges = @"
        using System;
        using System.Runtime.InteropServices;

         public class TokenAdjuster
         {
          [DllImport("advapi32.dll", ExactSpelling = true, SetLastError = true)]
          internal static extern bool AdjustTokenPrivileges(IntPtr htok, bool disall,
          ref TokPriv1Luid newst, int len, IntPtr prev, IntPtr relen);
          [DllImport("kernel32.dll", ExactSpelling = true)]
          internal static extern IntPtr GetCurrentProcess();
          [DllImport("advapi32.dll", ExactSpelling = true, SetLastError = true)]
          internal static extern bool OpenProcessToken(IntPtr h, int acc, ref IntPtr
          phtok);
          [DllImport("advapi32.dll", SetLastError = true)]
          internal static extern bool LookupPrivilegeValue(string host, string name,
          ref long pluid);
          [StructLayout(LayoutKind.Sequential, Pack = 1)]
          internal struct TokPriv1Luid
          {
           public int Count;
           public long Luid;
           public int Attr;
          }
          internal const int SE_PRIVILEGE_DISABLED = 0x00000000;
          internal const int SE_PRIVILEGE_ENABLED = 0x00000002;
          internal const int TOKEN_QUERY = 0x00000008;
```

```
              internal const int TOKEN_ADJUST_PRIVILEGES = 0x00000020;
              public static bool AddPrivilege(string privilege)
              {
               try
               {
                bool retVal;
                TokPriv1Luid tp;
                IntPtr hproc = GetCurrentProcess();
                IntPtr htok = IntPtr.Zero;
                retVal = OpenProcessToken(hproc, TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, ref htok);
                tp.Count = 1;
                tp.Luid = 0;
                tp.Attr = SE_PRIVILEGE_ENABLED;
                retVal = LookupPrivilegeValue(null, privilege, ref tp.Luid);
                retVal = AdjustTokenPrivileges(htok, false, ref tp, 0, IntPtr.Zero, IntPtr.Zero);
                return retVal;
               }
               catch (Exception ex)
               {
                throw ex;
               }
              }
              public static bool RemovePrivilege(string privilege)
              {
               try
               {
                bool retVal;
                TokPriv1Luid tp;
                IntPtr hproc = GetCurrentProcess();
                IntPtr htok = IntPtr.Zero;
                retVal = OpenProcessToken(hproc, TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, ref htok);
                tp.Count = 1;
                tp.Luid = 0;
                tp.Attr = SE_PRIVILEGE_DISABLED;
                retVal = LookupPrivilegeValue(null, privilege, ref tp.Luid);
                retVal = AdjustTokenPrivileges(htok, false, ref tp, 0, IntPtr.Zero, IntPtr.Zero);
                return retVal;
               }
               catch (Exception ex)
               {
                throw ex;
               }
              }
             }
"@
            Add-Type $AdjustTokenPrivileges
        }

        #Activate necessary admin privileges to make changes without NTFS perms
        [void][TokenAdjuster]::AddPrivilege("SeRestorePrivilege") #Necessary to set Owner
Permissions
        [void][TokenAdjuster]::AddPrivilege("SeBackupPrivilege") #Necessary to bypass Traverse
Checking
        [void][TokenAdjuster]::AddPrivilege("SeTakeOwnershipPrivilege") #Necessary to override
FilePermissions
    }
    Process {
        ForEach ($Item in $Path) {
            Write-Verbose "FullName: $Item"
            #The ACL objects do not like being used more than once, so re-create them on the
Process block
            $DirOwner = New-Object System.Security.AccessControl.DirectorySecurity
            $DirOwner.SetOwner([System.Security.Principal.NTAccount]$Account)
            $FileOwner = New-Object System.Security.AccessControl.FileSecurity
            $FileOwner.SetOwner([System.Security.Principal.NTAccount]$Account)
            $DirAdminAcl = New-Object System.Security.AccessControl.DirectorySecurity
            $FileAdminAcl = New-Object System.Security.AccessControl.DirectorySecurity
            $AdminACL = New-Object
System.Security.AccessControl.FileSystemAccessRule('Builtin\Administrators','FullControl','Containe
rInherit,ObjectInherit','InheritOnly','Allow')
            $FileAdminAcl.AddAccessRule($AdminACL)
            $DirAdminAcl.AddAccessRule($AdminACL)
            Try {
                $Item = Get-Item -LiteralPath $Item -Force -ErrorAction Stop
                If (-NOT $Item.PSIsContainer) {
                    If ($PSCmdlet.ShouldProcess($Item, 'Set File Owner')) {
                        Try {
```

```
                            $Item.SetAccessControl($FileOwner)
                        } Catch {
                            Write-Warning "Couldn't take ownership of $($Item.FullName)! Taking
FullControl of $($Item.Directory.FullName)"
                            $Item.Directory.SetAccessControl($FileAdminAcl)
                            $Item.SetAccessControl($FileOwner)
                        }
                    }
                } Else {
                    If ($PSCmdlet.ShouldProcess($Item, 'Set Directory Owner')) {
                        Try {
                            $Item.SetAccessControl($DirOwner)
                        } Catch {
                            Write-Warning "Couldn't take ownership of $($Item.FullName)! Taking
FullControl of $($Item.Parent.FullName)"
                            $Item.Parent.SetAccessControl($DirAdminAcl)
                            $Item.SetAccessControl($DirOwner)
                        }
                    }
                    If ($Recurse) {
                        [void]$PSBoundParameters.Remove('Path')
                        Get-ChildItem $Item -Force | Set-Owner @PSBoundParameters
                    }
                }
            } Catch {
                Write-Warning "$($Item): $($_.Exception.Message)"
            }
        }
    }
    End {
        #Remove priviledges that had been granted
        [void][TokenAdjuster]::RemovePrivilege("SeRestorePrivilege")
        [void][TokenAdjuster]::RemovePrivilege("SeBackupPrivilege")
        [void][TokenAdjuster]::RemovePrivilege("SeTakeOwnershipPrivilege")
    }
}
```