PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 12 August 2014

This Month's Topic:

Security-Related Tasks

Table of Contents

This issue compiles security-related code snippets from our daily tips column. Please note that this document by no means aims to be a complete coverage. It is most definitely not, yet contains many very useful code snippets.

If you have snippets of your own that you are willing to share, you are most welcome. We will update our issues regularly and are happy to include your feedback. Send your snippets to tobias@powertheshell.com.

1. Understanding Execution Policy

Out of the box, PowerShell won't run scripts. Whether you can run a script is governed by the "Windows PowerShell execution policy". The currently active policy can be retrieved like this:

```
PS> Get-ExecutionPolicy
Restricted
```

End users should use the setting “RemoteSigned”. It will allow you to run local scripts but will not allow scripts from outside the network domain, or downloaded scripts from the Internet (except if the script has a valid digital signature which most scripts won’t have).

Professional scripters can use “Bypass” which will allow you to run any script, regardless of location. “Unrestricted” in contrast will also let you run all scripts but will pop up a confirmation each time you are trying to run a script from a “remote” location.

Here is the line to change the execution policy for your own user account:

```
PS> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Bypass -Force

PS> Get-ExecutionPolicy
Bypass
```

To better understand execution policy, run this line:

```
PS> Get-ExecutionPolicy -List

                                 Scope                          ExecutionPolicy
                                 -----                          ---------------
                         MachinePolicy                                Undefined
                            UserPolicy                                Undefined
                               Process                                Undefined
                           CurrentUser                                   Bypass
                          LocalMachine                             RemoteSigned
```

As you can see, there are five execution policies. PowerShell goes from top to down, and the first one that is not “Undefined” becomes the effective execution policy. If all are “Undefined”, then the resulting policy is “Restricted” – that is the default behavior on new systems. Only Windows Server 2012 R2 defaults to “RemoteSigned”

The scopes “MachinePolicy” and “UserPolicy” are set by Active Directory group policies. If these are set, then they override all policies below.

The scope “Process” is valid only in the current PowerShell session. This is the policy that gets set when you run powershell.exe with the -ExecutionPolicy parameter.

2. Overriding Execution Policy

Execution policy is not a security boundary to protect you from evil. It is just a seat belt to protect you from yourself. That’s why there are plenty of ways to override the execution policy and execute PowerShell commands.

This approach would read a script and pipe it to powershell.exe directly (don’t forget the “-“ at the end of the command line!):

```
Get-Content ‘C:\somescript.ps1’ -Raw | powershell.exe -noprofile -
```

3. Listing NTFS Permissions

To view NTFS permissions for folders or files, use Get-Acl. It won’t show you the actual permissions at first, but you can make them visible like this:

```
Get-Acl -Path $env:windir |
  Select-Object -ExpandProperty Access
```

4. Reading Security Descriptors in SDDL Form

Security descriptors work like electronic locks. They secure securable objects, such as files and folders on an NTFS drive, or registry keys, or Active Directory objects. Security descriptors typically are complex objects but can be flattened into a string representation. This string representation is called “SDDL” (Security Descriptor Definition Language), and PowerShell can display any security descriptor in SDDL form.

This outputs the SDDL for a file (make sure the file exists or specify a different file or folder):

```
$DogACL = Get-Acl -Path c:\test\dog.txt
$DogSDDL = $DogACL.GetSecurityDescriptorSddlForm('All')
$DogSDDL
```

The same code could display the SDDL for a registry key. Just change the path:

```
$DogACL = Get-Acl -Path HKCU:\Software
$DogSDDL = $DogACL.GetSecurityDescriptorSddlForm('All')
$DogSDDL
```

5. Clone NTFS Permissions

NTFS access permissions can be complex and tricky. To quickly assign NTFS permissions to a new folder, you can simply clone permissions from another folder that you know has the correct permissions applied.

Here is the first part. This creates a sample folder in your temporary folder, then opens the temporary folder and selects the newly created folder for you:

```
$OriginalPath = "$env:temp\sample"

New-Item -Path $OriginalPath -ItemType Directory

# manually assign correct permissions to folder "sample"
Explorer.exe "/Select,$OriginalPath"
```

Right-click the "sample" folder, choose "Properties", then click on the "Security" tab. Now, add the security permissions you need and apply them.

When your "prototype" folder is correctly configured, use this code to read the security information from it:

```
$sddl = (Get-Acl $OriginalPath).Sddl
```

When you look at $sddl, it will contain a long text string that represents all the security information you just added to the folder.

From this point on, you do not need the prototype folder any more. It was needed only to create the SDDL definition string. You can now apply (clone) the security information to any other folder you want:

```
$newpath = "$env:temp\newfolder"
md $newpath
$sd = Get-Acl -Path $newpath
$sd.SetSecurityDescriptorSddlForm($sddl)
Set-Acl -Path $newpath -AclObject $sd
```

Unfortunately, setting ACLs this way always requires administrator privileges.

Note that you can apply this technique to any valid PowerShell path. So you can apply it to registry keys as well. Use the very same pieces of example code, and replace the file system paths with registry paths (like HKCU:\Software\SomeKey).

6. Adding Permissions

To add new permissions to an existing security descriptor, create an appropriate AccessRule object and configure it.

This script adds a new FileSystemAccessRule to the security descriptor of a file, granting read and write access to mydomain\myaccount.

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He organizes the annual German PowerShell Conference, has published four PowerShell books (MS Press Germany) and is a regular speaker at conferences such as the PowerShell Summit in Amsterdam. As a developer and software architect, Weltner recently published the commercial PowerShell ISE extension "ISESteroids", turning the built-in PowerShell editor into a sophisticated development environment (http://www.powertheshell.com/isesteroids/).

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

Make sure you adjust both user account and filename before you test the code:

```
$colRights = [System.Security.AccessControl.FileSystemRights]'Read, Write'
$InheritanceFlag = [System.Security.AccessControl.InheritanceFlags]::None
$PropagationFlag = [System.Security.AccessControl.PropagationFlags]::None
$objType =[System.Security.AccessControl.AccessControlType]::Allow
$objUser = New-Object System.Security.Principal.NTAccount('mydomain\myaccount')
$objACE = New-Object System.Security.AccessControl.FileSystemAccessRule `
    ($objUser, $colRights, $InheritanceFlag, $PropagationFlag, $objType)

# get original SD
$catACL = Get-Acl 'C:\test\cat.txt'

# add permission
$catACL.AddAccessRule($objACE)

# write back the appended SD
Set-Acl 'C:\test\cat.txt' $catACL
```

You can use the same technique for other PowerShell drives, such as the registry. Just make sure you create the appropriate AccessRule object. To add permissions to registry keys, use a RegistryAccessRule object instead of a FileSystemAccessRule object.

7. Removing Permissions

To selectively remove a permission from a security descriptor, get access to the access control entries, pick the ones to remove, and then write back the changed security descriptor.

This example removes all permissions and denials for the account mydomain\myaccount. Make sure you adjust both account and filename before you test the code.

```
$catACL = Get-Acl c:\test\cat.txt

$unwanted = $catACL.Access |
  Where-Object { $_.IdentityReference.Value -eq 'mydomain\myaccount' }

$unwanted | ForEach-Object { $null = $catACL.RemoveAccessRule($_) }

Set-Acl -Path c:\test\cat.txt -AclObject $catACL
```

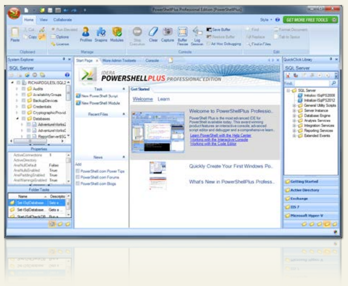
8. Checking Administrator Privileges

There are numerous ways to find out if a script runs elevated. Here's a pretty simple approach that uses whoami.exe (ships with Win7/Server 200 R2 or better):

```
(whoami.exe /all | Select-String S-1-16-12288) -ne $null
```

If you do not have whoami.exe, or if you are looking for a more integrated approach, you can use a line that is a little longer but identifies Admin status directly, without calling an external program:

```
(New-Object System.Security.Principal.WindowsPrincipal([System.Security.
Principal.WindowsIdentity]::GetCurrent())).IsInRole([System.Security.Principal.
WindowsBuiltInRole]::Administrator)
```



You can make this a function, too:

```
function Test-Admin
{
  $wid = [System.Security.Principal.WindowsIdentity]::GetCurrent()
  $prp = New-Object System.Security.Principal.WindowsPrincipal($wid)
  $adm = [System.Security.Principal.WindowsBuiltInRole]::Administrator
  $prp.IsInRole($adm)
}
```

9. Show Admin Status in Prompt

If you want to know whether your PowerShell host has currently full Administrator privileges, here is an (admittedly long) one-liner that lets you know:

```
(New-Object System.Security.Principal.WindowsPrincipal([System.Security.
Principal.WindowsIdentity]::GetCurrent())).IsInRole([System.Security.Principal.
WindowsBuiltInRole]::Administrator)
```

Try executing this line in a regular PowerShell and then in an elevated shell, and check out the difference. Or, create your own console prompt which turns red when you have admin privileges:

```
function prompt {
  if ((New-Object System.Security.Principal.WindowsPrincipal([System.Security.
Principal.WindowsIdentity]::GetCurrent())).IsInRole([System.Security.Principal.
WindowsBuiltInRole]::Administrator)) {
    Write-Host '(Admin)' -ForegroundColor Red -NoNewLine
    $user = 'Administrator: '
  } else {
    $user = ''
  }
  'PS> '
  $Host.UI.RawUI.WindowTitle = "$user $(Get-Location)"
}
```

10. Waiting for Elevated Processes

You can easily run a process elevated (with Administrator privileges), but if you do, you may not be able to wait for the process to complete.

Here is an example that you can try:

```
# launch Notepad elevated
Start-Process -FilePath notepad.exe -Verb Runas -WorkingDirectory c:\

# launch Notepad elevated and wait for it (fails in PowerShell 2.0)
Start-Process -FilePath notepad.exe -Verb Runas -WorkingDirectory c:\ -Wait
```

The second call will fail in PowerShell 2.0 when your calling shell was not elevated already. This is because in PowerShell 2.0, a non-elevated process can only read very limited information from an elevated process.

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the co-founder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

11. Executing Elevated PowerShell Code

Sometimes, a script may want to execute some portion of its code elevated, for example to write to HKLM in the Registry or change protected settings. Instead of requiring the user to run the entire script elevated, you can execute portions of it in a separate PowerShell that gets elevated automatically--provided that your user account owns Administrator privileges, or you can provide logon credentials for such an account.

This line, for example, will create a Registry key in the protected HKLM hive. If the script runs elevated already, the key will just be created. If the script has no special privileges, a UAC elevation prompt appears and elevates the script portion that needs it:

```
$regkeyPath = 'HKLM:\Software\NewKey1'

# test registry key
Test-Path $regkeyPath

# code to create registry key
$code = "New-Item $regkeyPath"

# run code elevated and wait for the process (requires PowerShell 3.0, remove -Wait
for older versions)
$process = Start-Process -FilePath powershell.exe -Verb Runas -WindowStyle Minimized
-ArgumentList "-noprofile -command $code" -WorkingDirectory c:\ -Wait

# test registry key again
Test-Path $regkeyPath
```

Likewise, your code could execute any other piece of code that needs Administrator privileges, for example restart a service or write to a protected file location.

12. Requiring Administrator Privileges

If you know that a given script requires Administrator privileges, then you can mark this script with a special requirement. This will cause the entire script to fail if it was run from a non-elevated user.

This example will run if called by a user with Administrator privileges, and fail for anyone else. Note that the "requires" statement will not elevated the script.

```
#Requires -RunAsAdministrator
'I am Admin!'
```

Note also that this statement will only work in saved scripts. It will not work in "Untitled" scripts.

13. Automatically Elevating Script

If you know that a script needs Administrator privileges, then you can use this code to auto-elevate the script if it is run by a non-administrator user:

```
$identity = [Security.Principal.WindowsIdentity]::GetCurrent()
$principal = New-Object Security.Principal.WindowsPrincipal $identity
$isAdmin = $principal.IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)

if ($isAdmin -eq $false)
{
  $Script = $MyInvocation.MyCommand.Path
  $Args = '-noprofile -nologo -executionpolicy bypass -file "{0}"' -f $Script

  Start-Process -FilePath 'powershell.exe' -ArgumentList $Args -Verb RunAs
  exit
}

'Running with Admin Privileges'
Read-Host 'PRESS ENTER'
```

14. Asking for Credentials

When you write functions that accept credentials as parameters, add a transformation attribute! This way, the user can either submit a credential object (for example, a credential supplied from Get-Credential), or simply a user name as string.

The transformation attribute will then convert this string automagically into a credential and ask for a password.

Here is a sample function:

```
function do-something  {
  param(
    [System.Management.Automation.Credential()]
    $Credential
  )
  ‘$Credential now holds a valid credential object’
  $Credential
}
```

When you run do-something without parameters, it will automatically invoke the credentials dialog.

15. Finding Expired Certificates

To find expired certificates, you can browse the cert: drive and look for certificates that have a property NotAfter that lies in the past:

```
$today = Get-Date


Get-ChildItem -Path cert:\ -Recurse |
  Where-Object { $_.NotAfter } |
  Where-Object { $_.NotAfter -lt $today } |
  Select-Object -Property Subject, PSPath, NotAfter |
  Out-GridView
```

16. Deleting Installed Certificates

If you want to permanently delete a digital certificate in your certificate store, here is how. This line lists all your personal certificates:

```
Get-ChildItem cert:\CurrentUser\My
```

Let’s assume there is a certificate with a thumbprint like “5F7EAAA46548F83742D68B06148E71D4AB40A293” (replace with the thumbprint of the actual certificate you want to get rid of).

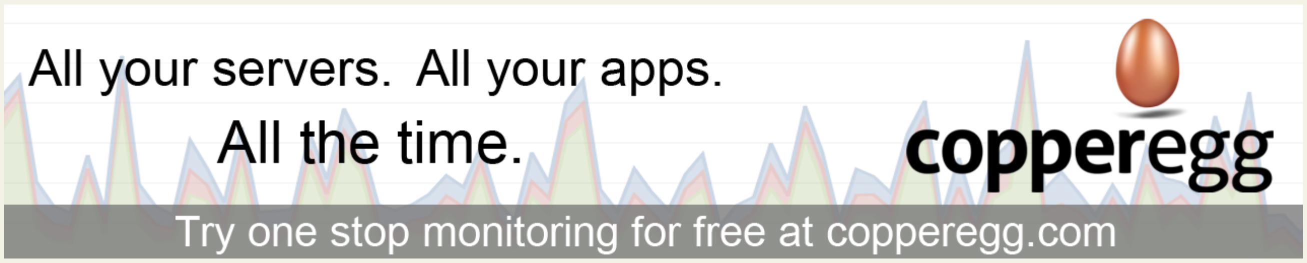
Here is how you can get and/or delete the certificate (again, make sure you change the thumbprint ID in the example to the one of the certificate you want to target on your machine):

```
# get certificate
Get-Item -Path cert:\CurrentUser\My\5F7EAAA46548F83742D68B06148E71D4AB40A293

# delete certificate (WARNING: THIS WILL REMOVE THE CERTIFICATE! REQUIRES POWERSHELL 3.0)
Remove-Item -Path cert:\CurrentUser\My\5F7EAAA46548F83742D68B06148E71D4AB40A293
```

Note that Remove-Item will not be able to delete a certificate in PowerShell 2.0. Here, you would have to resort to a low-level approach:

```
$cert = Get-Item -Path cert:\CurrentUser\My\5F7EAAA46548F83742D68B06148E71D4AB40A293
$store = Get-Item $cert.PSParentPath
$store.Open(‘ReadWrite’)
$store.Remove($cert)
$store.Close()
```


All your servers. All your apps.
All the time.
copperegg
Try one stop monitoring for free at copperegg.com

17. Turning User Names into SIDs

If you want to translate a valid user name to its security identifier (SID), here is a function to do that for you:

```
function Convert-Name2SID
{
  param
  (
    [System.Object]
    $Name,
    [System.Object]
    $Domain = $env:userdomain
  )

  $objUser = New-Object System.Security.Principal.NTAccount($domain, $name)
  $strSID = $objUser.Translate([System.Security.Principal.SecurityIdentifier])
  $strSID.Value
}
```

Try it with your current user name:

```
PS> Convert-Name2SID -Name $env:USERNAME
S-1-5-21-1907506615-3936657230-2684137421-1001
```

18. Turning SIDs into Real Names

If you just know the raw security identifier (SID), you can use the following function to translate it into real names:

```
function Convert-SID2User
{
  param
  (
    [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
    [String]
    $SID
  )

  process
  {
    $objSID = New-Object System.Security.Principal.SecurityIdentifier($SID)
    try {
      $objUser = $objSID.Translate( [System.Security.Principal.NTAccount])
      $objUser.Value
    } catch { $SID }
  }
}
```

And here is a quick example on how to use it:

```
PS> Convert-SID2User -SID S-1-5-32-544
BUILTIN\Administrators

PS>
```

As you see, Convert-SID2User actually returns Identities (not just users but also groups, as you can see in this example).

19. Finding Profiles

To get a list of user profiles present on your machine, you can query the registry:

```
function Get-Profile
{
  $key = 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList'
  Get-ChildItem $key -Name
}

Get-Profile
```

The result is a list of SIDs. With the Convert-SID2User function described before, you can now translate the SIDs to real names:

```
# translate SIDs to user names
# (requires the Convert-SID2User function covered before)
Get-Profile | Convert-SID2User
```

20. Finding Security Info for the Current User

With a single call to a system function, you can get a lot of security-related information about the currently logged on user (the user that is running the code):

```
PS> [System.Security.Principal.WindowsIdentity]::GetCurrent()


AuthenticationType : LiveSSP
ImpersonationLevel : None
IsAuthenticated    : True
IsGuest            : False
IsSystem           : False
IsAnonymous        : False
Name               : TOBI2\Tobias
Owner              : S-1-5-21-1907506615-3936657230-2684137421-1001
User               : S-1-5-21-1907506615-3936657230-2684137421-1001
Groups             : {S-1-1-0, S-1-5-21-1907506615-3936657230-2684137421-1002,
                     S-1-5-32-559, S-1-5-32-545...}
Token              : 4892
UserClaims         : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nam
                     e: TOBI2\Tobias, http://schemas.microsoft.com/ws/2008/06/i
                     dentity/claims/primarysid:
                     S-1-5-21-1907506615-3936657230-2684137421-1001, http://sch
                     emas.microsoft.com/ws/2008/06/identity/claims/groupsid:
                     S-1-1-0, http://schemas.xmlsoap.org/ws/2005/05/identity/cl
                     aims/denyonlysid: S-1-5-114...}
DeviceClaims       : {}
Claims             : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nam
                     e: TOBI2\Tobias, http://schemas.microsoft.com/ws/2008/06/i
                     dentity/claims/primarysid:
                     S-1-5-21-1907506615-3936657230-2684137421-1001, http://sch
                     emas.microsoft.com/ws/2008/06/identity/claims/groupsid:
                     S-1-1-0, http://schemas.xmlsoap.org/ws/2005/05/identity/cl
                     aims/denyonlysid: S-1-5-114...}
Actor              :
BootstrapContext   :
Label              :
NameClaimType      : http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
RoleClaimType      : http://schemas.microsoft.com/ws/2008/06/identity/claims/gr
                     oupsid
```

To find out the user SID, for example, try this:

```
PS> [System.Security.Principal.WindowsIdentity]::GetCurrent().User.Value
S-1-5-21-1907506615-3936657230-2684137421-1001
```

21. Finding Current Group Memberships

To find out the groups your user belongs to, you would typically query the Active Directory. There is a much easier way, though: take a look at the user access token. This token already contains all group memberships – including nested group memberships:

```
PS> [System.Security.Principal.WindowsIdentity]::GetCurrent().Groups | Select-Object
-ExpandProperty Value

S-1-1-0
S-1-5-21-1907506615-3936657230-2684137421-1002
S-1-5-32-559
S-1-5-32-545
S-1-5-4
S-1-2-1
S-1-5-11
S-1-5-15
S-1-11-96-3623454863-58364-18864-2661722203-1597581903-930530977-1207164330-214
1635683-1414797606-1103398013
S-1-5-113
S-1-2-0
S-1-5-64-32
```

You will get back the SIDs of all groups you are in. To translate the SIDs to real group names, use the Convert-SID2User function covered before:

```
[System.Security.Principal.WindowsIdentity]::GetCurrent().Groups |
  Select-Object -ExpandProperty Value |
  Convert-SID2User
```

You could use this approach in logon scripts to check whether a user is member of a particular group.

22. Generate Random Passwords

Here is a line that can create simple random passwords. Simply put the characters that you would allow in your passwords into the string. Omit characters that can be confusing, like "0OoQ" or "1lI".

```
$allowed = 'abcdefghiklmnprstuvwxyzABCDEFGHKLMNPRSTUVWXYZ123456789!"$%&=#*'
-join (Get-Random -InputObject $allowed.ToCharArray() -Count 20)
```

23. Encrypt Files with EFS

EFS (Encrypting File System) makes sure that a file can only be read by the person that encrypted it. This can be a quick and easy way for you to store sensitive information like passwords. The following code would use PowerShell to encrypt a file. This will work only if EFS is enabled on your system.

```
$file = Get-Item -Path c:\somefile.txt
$file.Encrypt()

# to decrypt, use this:
#$file.Decrypt()
```

If a file was successfully encrypted by EFS, it will turn green in the File Explorer.

24. Entering Passwords Securely

You can use Read-Host with the -AsSecureString parameter to enter a password with hidden characters:

```
$password = Read-Host -AsSecureString 'Password'
```

This will return a secure string. To convert the secure string back to a plain string, use this:

```
$passwordPlain = (New-Object System.Management.Automation.PSCredential(
'dummy',$password)).GetNetworkCredential().Password
```

So this would be a function you can use to ask for a hidden password and use the plain text entered in your script:

```
function Get-PasswordString
{
  param
  (
    $Prompt = 'Enter Password'
  )

  $password = Read-Host -AsSecureString $Prompt
  (New-Object System.Management.Automation.PSCredential('dummy',$password)).
GetNetworkCredential().password
}
```

25. Decrypting SecureStrings

SecureStrings are encrypted text strings. The encryption helps protect the string content if it has to travel to another computer (i.e. network passwords). It is obviously not protecting the string from the one who encrypted it in the first place.

Here is code to proof this: it asks for a SecureString (similar to the previous example), and then uses a system function to decrypt it again:

```
$SecureString = Read-Host -AsSecureString 'Secret'
[Runtime.InteropServices.Marshal]::PtrToStringAuto(
[Runtime.InteropServices.Marshal]::SecureStringToBSTR( $SecureString))
```

26. Unattended Authentication

Typically, when you use credentials to log on as someone else, PowerShell cmdlets will open a dialog where you can enter your username and password.

To run commands unattended, you can create a credential object yourself. Note however that this would require you to hard-code the password in your script which is a very risky technique.

This example would run Notepad as "companydomain\user1" without popping up a credentials dialog:

```
$user = 'companydomain\User1'
$password = 'P@ssw0rd1' | ConvertTo-SecureString -AsPlainText -Force

$cred = New-Object System.Management.Automation.PSCredential($user, $password)

Start-Process notepad.exe -Credential $cred -LoadUserProfile
```