

Manipulación de la base de datos de cuentas local

Debe saber que el soporte de Active Directory Servicio Interfaces se podría mejorar bastante. Prácticamente no ha evolucionado entre las versiones 1 y 2 de PowerShell. Sepa que ADSI forma parte de una tecnología desfasada. A pesar de todo, siempre que se trata de administrar una de base de datos de cuentas local, no tendrá otra posibilidad para manipular los objetos ADSI.

Como vamos a ver, no existe ningún conjunto de comandos PowerShell dedicado, y a algunas propiedades y métodos de objetos se accede de maneras a veces un poco... «inusuales». Es decir que deberemos por el momento acceder a los objetos en bruto de PowerShell.

Ya sea en un equipo que ejecute un sistema operativo cliente (como Windows Vista, XP o anterior) o en un sistema operativo servidor (a excepción del Controlador de dominio) existe siempre una base de datos de cuenta local. Ésta, también denominada «base de datos SAM» (*Security Account Manager*), contiene objetos de tipo usuarios, y grupos de usuarios, al igual que sus respectivos sistemas identificativos: el SID (*Security Identifier*).

Para acceder a un objeto de servicio de directorio de tipo SAM o AD DS, vamos a utilizar el atajo [ADSI]. Lo que cambiará, es únicamente el proveedor de servicios al que llamaremos. Para la manipulación de la base de datos SAM se tratará del proveedor «WinNT:».

Destacar que las mayúsculas y minúsculas tienen aquí una gran importancia. Esto es una condición impuesta no por PowerShell sino por ADSI y sea cual sea el lenguaje de script utilizado.

1. Administración de los grupos

a. Listar los grupos

Para empezar, listemos los grupos locales de una máquina local o remota, con el pequeño script siguiente:

```
# Get-LocalGroups.ps1
param([String]$maquina='.')
$conexion = [ADSI]"WinNT://$maquina"
$conexion.PSBase.Children |
    Where {$_.PSBase.SchemaClassName -eq 'group'} | foreach{$_.Name}
```

Recuperamos el valor del parámetro «-maquina» en la variable `$maquina`. Si no la definimos, el valor de esta última se inicializará con el valor «.». Creamos seguidamente una conexión a la base de datos SAM local gracias a [ADSI] "WinNT://\$maquina" después aplicamos un filtro para únicamente recuperar los objetos de tipo grupo. Finalmente mostramos la propiedad `Name` de los objetos filtrados.

Ejemplo de utilización en una máquina Windows Vista:

```
PS > ./Get-LocalGroups.ps1

Administradores
Duplicadores
IIS_IUSRS
Invitados
Lectores del registro de eventos
Operadores criptográficos
Operadores de configuración de red
Operadores de copia de seguridad
Usuarios
Usuarios avanzados
Usuarios COM distribuidos
```

```
Usuarios de escritorio remoto
Usuarios del monitor de sistema
Usuarios del registro de rendimiento
HelpLibraryUpdaters
```



Debe ser administrador de la máquina para que el script funcione.

b. Listar los miembros de un grupo

He aquí el script que nos permitirá realizar esta tarea:

```
# Get-LocalGroupMembers.ps1
param ([String]$maquina='.',
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'))

$conexion = [ADSI]"WinNT://$maquina/$grupo,group"
$conexion.psbase.invoke('Members') |
foreach{$_.GetType().InvokeMember('Name', 'GetProperty', $null, $_, $null)}
```

Ejemplo de utilización:

```
PS > ./Get-LocalGroupMembers.ps1 -grupo Administradores

Administrador
Admins de dominio
Oscar
Roberto
```

Aunque hemos podido llegar a listar los nombres de los miembros, esto podría resultarnos en algunos casos insuficiente. Por ejemplo, podríamos tener en nuestro grupo miembros pertenecientes a un dominio.

Es la razón por la que vamos a modificar nuestro script para obtener, además del nombre, la valiosa información que constituye el dominio. Para ello vamos a consultar la propiedad `AdsPath` en lugar de `Name`.

```
# Get-LocalGroupMembersV2.ps1
param ([String]$maquina='.',
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'))

$conexion = [ADSI]"WinNT://$maquina/$grupo,group"
$conexion.psbase.invoke('Members') |
%{$_.GetType().InvokeMember('AdsPath', 'GetProperty', $null, $_, $null)}
```

Probemos nuestro nuevo script:

```
PS > ./Get-LocalGroupMembersV2.ps1 -grupo Administradores

WinNT://PCVISTA/Administrador
WinNT://PCVISTA/Roberto
WinNT://PS-SCRIPTING/Admins de dominio
WinNT://PS-SCRIPTING/Oscar
```

Hemos hecho bien en hacer la verificación ya que en este ejemplo `Administrador` y `Roberto` son usuarios locales de la máquina `PCVISTA`, mientras que `Admins de dominio` y `Oscar` son miembros del dominio `PS-SCRIPTING`.

Observe que la información devuelta no está en el formato más común. Normalmente se designa un usuario o un grupo mediante el formato dominio\usuario. Cuando se de el caso, PowerShell posee operadores de tratamiento de cadenas muy potentes que nos van a resolver este pequeño «problema»:

```
PS > ./Get-LocalGroupMembersV2.ps1 -grupo Administradores |  
foreach {($_ -replace 'WinNT://', '') -replace '/', '\'}  
  
PCVISTA\Administrador  
PCVISTA\Roberto  
PS-SCRIPTING\Admins de dominio  
PS-SCRIPTING\Oscar
```

Será suficiente enviar cada resultado a un `foreach`, quien por cada ocurrencia empezará por eliminar la cadena «WinNT://» reemplazándola por una cadena vacía y después reemplazará el carácter «/» por «\».

c. Añadir un miembro a un grupo


Podemos añadir a los grupos usuarios locales o usuarios del ámbito. Veamos el primer caso:

```
# Add-LocalGroupMember.ps1  
param ([String]$maquina='.',  
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'),  
      [String]$Usuario=$(Throw "¡Nombre del usuario obligatorio!"))  
  
$conexion = [ADSI]"WinNT://$maquina/$grupo,group"  
$conexion.Add("WinNT://$Usuario")
```

Sírvase señalar que no tenemos necesidad de pasar el nombre de la máquina local además del nombre de usuario al método `Add`; en efecto, ADSI entiende que se trata de un usuario local de la máquina.

Ejemplo de utilización:


```
PS > ./Add-LocalGroupMember.ps1 -grupo Administradores -usu Oscar
```

 No estamos obligados a especificar los parámetros en su totalidad mientras no haya ninguna ambigüedad. Como prueba hemos utilizado `-usu` en lugar de `-usuario`. Si en cambio, tuviésemos un segundo parámetro denominado por ejemplo `-usufructo`, entonces PowerShell no sabría a qué parámetro asociar la abreviación `-usu` y por tanto nos enviaría un error.

Adición de un miembro de dominio

También podemos añadir una cuenta del dominio a un grupo local. Para esto debemos utilizar el comando siguiente:

```
PS > .\Add-LocalGroupMember.ps1 -g Administradores -u 'Dominio/Usuario'
```

 Preste atención a la dirección de la barra del separador Dominio/Usuario. Nuestra función utiliza el proveedor de servicio «WinNT://» y habrá observado que este último sólo trabaja con slash y no con anti-slash («\»).

Dejamos en sus manos la tarea de modificar el script `Add-LocalGroupMember.ps1` para cambiar la forma de añadir los miembros. Un miembro podrá ser tanto un usuario, como un grupo global.

d. Eliminar un miembro de un grupo

Para añadir un miembro a un grupo hemos usado el método `Add`; para eliminar un miembro, utilizaremos simplemente el método `Remove`.

Así nuestro script anterior pasará a ser:

```
# Remove-LocalGroupMember.ps1
param ([String]$maquina='.',
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'),
      [String]$Usuario=$(Throw "¡Nombre del usuario obligatorio!"))

$conexion = [ADSI]"WinNT://$maquina/$grupo,group"
$conexion.Remove("WinNT://$Usuario")
```

Ejemplo de utilización:

```
PS > ./Remove-LocalGroupMember.ps1 -grupo Administradores -usu Roberto
```

Acabamos de eliminar el usuario local llamado «Roberto». Eliminemos ahora un miembro del dominio perteneciente al grupo `Administradores`:

```
PS > ./Remove-LocalGroupMember.ps1 -g Administradores -u 'Dominio/Usuario'
```

e. Crear un grupo

Hasta ahora hemos modificado grupos ya presentes en el sistema. Veamos ahora cómo crear un grupo.

```
# New-LocalGroup.ps1
param ([String]$maquina='.',
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'),
      [String]$Descripcion)

$conexion = [ADSI]"WinNT://$maquina"
$objGrupo = $conexion.Create('group', $grupo)
$objGrupo.Put('Description', $Descripcion)
$objGrupo.SetInfo()
```

El objeto grupo posee la propiedad `Description` que podemos modificar.

Ejemplo de utilización:

```
PS > ./New-LocalGroup.ps1 -g GrupoPrueba -desc 'Grupo de prueba'
```

No nos queda más que añadir miembros a través del script `Add-LocalGroupMember.ps1` que hemos creado anteriormente.

f. Eliminar un grupo

```
# Remove-LocalGroup.ps1
param ([String]$maquina='.',
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'))

$conexion = [ADSI]"WinNT://$maquina"
$conexion.Delete('group', $grupo)
```

Ejemplo de utilización:

```
PS > ./Remove-LocalGroup.ps1 -grupo GrupoPrueba
```

g. Modificar un grupo

Cuando hemos creado un grupo le hemos añadido también una descripción que modifica el valor del atributo `Description` gracias al método `Put` seguido de `SetInfo`.

`SetInfo` permite validar una modificación. Si omite `SetInfo` cuando modifique las propiedades de un objeto, el sistema no tendrá en cuenta sus modificaciones.



Existe otro método que produce los mismos efectos que `setInfo`. Se trata de `commitChanges`. La única diferencia es que `commitChanges` se aplica a un objeto de tipo `psobject` y en particular a la propiedad `psBase`. A lo largo de los muchos ejemplos que veremos en adelante utilizaremos indistintamente uno u otro método.

En un grupo, podremos modificar dos cosas:

- su nombre,
- su descripción.

Veamos como renombrar un grupo:

```
# Rename-LocalGroup.ps1
param ([String]$maquina='.',
      [String]$OldName=$(Throw '¡Nombre del grupo obligatorio!'),
      [String]$NewName=$(Throw "¡Nombre de un grupo obligatorio!"))

$conexion = [ADSI]"WinNT://$maquina/$OldName,group"
$conexion.psbase.Rename($NewName)
$conexion.SetInfo()
```

Y ahora como modificar su descripción:

```
# Set-LocalGroupDescription.ps1
param ([String]$maquina='.',
      [String]$Grupo=$(Throw '¡Nombre del grupo obligatorio!'),
      [String]$Descripcion)

$conexion = [ADSI]"WinNT://$maquina/$grupo,group"
$conexion.put('Description',$Descripcion)
$conexion.SetInfo()
```

Ejemplo de utilización:

```
PS > ./Set-LocalGroupDescription.ps1 -g GrupoPrueba -d 'Nueva descripción'
```

2. Administración de los usuarios locales

a. Listar los usuarios

La obtención de la lista de usuarios de la base de datos SAM se basa en el mismo principio que el del script que nos devuelve la lista de los grupos. Basta con modificar el filtro a la clase `SchemaClassName`:

```
# Get-LocalUsers.ps1
param ([String]$maquina='.')

$conexion = [ADSI]"WinNT://$maquina"
$conexion.PSBase.Children |
Where {$_.PSBase.SchemaClassName -eq 'user'} | Foreach{$_.Name}
```

Ejemplo de utilización:

```
PS > ./Get-LocalUsers.ps1

Administrador
Invitado
Oscar
Roberto
```

Para listar las cuentas de una máquina remota:

```
PS > ./Get-LocalUsers.ps1 -maquina MiMaquinaRemota

Administrador
HelpAssistant
Invitado
SUPPORT_388945a0
```

b. Crear un usuario local

La creación de un usuario se realiza con la ayuda del método `Create`; al que le pasaremos como primer argumento «user», seguido del nombre. Aprovecharemos también para definir una contraseña gracias a `SetPassword`. Por último, terminaremos por añadir una descripción opcional.

```
# New-LocalUser.ps1
param ([String]$maquina='.',
      [String]$Nombre=$(Throw ";Nombre de usuario obligatorio!"),
      [String]$Contraseña=$(Throw ";Contraseña obligatoria!"),
      [String]$Descripcion)

$objMaquina = [ADSI]"WinNT://$maquina"
$objUser = $objMaquina.Create('user', $Nombre)
$objUser.SetPassword($Contraseña)
$objUser.psbase.InvokeSet('Description', $Descripcion)

$objUser.SetInfo()
```

Ejemplo de utilización:

```
PS > ./New-LocalUser.ps1 -nom Javier -Contraseña 'P@ssw0rd'
```

Hubiésemos podido también añadir una descripción a nuestro usuario del siguiente modo:

```
PS > ./New-LocalUser.ps1 -nom Javier -Contraseña 'P@ssw0rd' -Desc 'abcd'
```

c. Modificar un usuario local

Es posible parametrizar muchas más propiedades que la contraseña y la descripción. Aquí tiene un extracto de las propiedades más habituales para la administración de los usuarios locales:

Propiedad	Tipo	Descripción
Description	String	Corresponde al campo Descripción.
FullName	String	Corresponde al campo Nombre completo.
UserFlags	Integer	Permite añadir el estado de la cuenta: desactivada, la contraseña nunca expira, etc.
HomeDirectory	String	Ruta del directorio base.
HomeDirDrive	String	Letra asociada a la ruta del directorio base.
Profile	String	Ubicación del perfil Windows.
LoginScript	String	Nombre del script de logon.
ObjectSID	String	SID del usuario.
PasswordAge	Time	Duración de la utilización de la contraseña actual en número de segundos después del último cambio de contraseña.
PasswordExpired	Integer	Indica si la contraseña ha expirado. Valdrá cero si la contraseña no ha expirado, u otro valor si ha expirado.
PrimaryGroupID	Integer	Identificador del grupo primario de pertenencia del usuario.

Para modificar una propiedad de esta lista, utilice la sintaxis siguiente:

```
$objUser.PSBase.InvokeSet('Propiedad', $Valor)
```

No olvide de utilizar el método `SetInfo` para validar la modificación.

Ejemplo:

```
...
PS > $objUser.PSBase.InvokeSet('Propiedad', $Valor)
PS > $objUser.SetInfo()
```

Acabamos de ver que existe un cierto número de propiedades sobre las que podemos actuar. Veamos cómo las descubriremos con PowerShell:

```
PS > $user=[ADSI]'WinNT://./Oscar,user'
PS > $user

distinguishedName
-----
```

Lamentablemente, las propiedades de nuestro usuario no están expuestas con el adaptador de tipo ADSI. Dicho esto, es posible ver una parte de las propiedades en bruto de nuestro objeto usuario gracias a la propiedad `PSAdapted`. Ahora, probemos lo siguiente:

```
PS > $user.PSAdapted

UserFlags           : {545}
MaxStorage          : {-1}
PasswordAge         : {20}
PasswordExpired     : {0}
LoginHours          : {255 255 255 255 255 255 255 255 255 255 255
                    255 255 255 255 255 255 255 255 255 255}
FullName            : {Oscar Vázquez}
Description          : {Cuenta de usuario}
BadPasswordAttempts : {0}
LastLogin           : {20/07/2009 15:09:53}
HomeDirectory       : {}
LoginScript         : {}
Profile             : {}
HomeDirDrive        : {}
Parameters          : {}
PrimaryGroupID      : {513}
Name                : {Oscar}
MinPasswordLength   : {0}
MaxPasswordAge      : {3628800}
MinPasswordAge      : {0}
PasswordHistoryLength : {0}
AutoUnlockInterval  : {1800}
LockoutObservationInterval : {1800}
MaxBadPasswordsAllowed : {0}
objectSid           : {1 5 0 0 0 0 0 5 21 0 0 0 124 224 91 123
                    165 226 143 247 33 244 133 79 232 3 0 0}
```

Probemos por ejemplo de obtener la fecha de la última conexión:

```
PS > $user.PSBase.InvokeGet('lastlogin')

viernes 20 julio 2009 15:09:53
```

Sólo por curiosidad veamos de qué tipo resulta ser esta información:

```
PS > ($user.PSBase.InvokeGet('lastlogin')).GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                     System.ValueType
```

La información es de tipo `DateTime`, lo que es perfecto si queremos formatear de nuevo toda esta información o realizar pruebas de comparación de fechas.

Activar/desactivar una cuenta

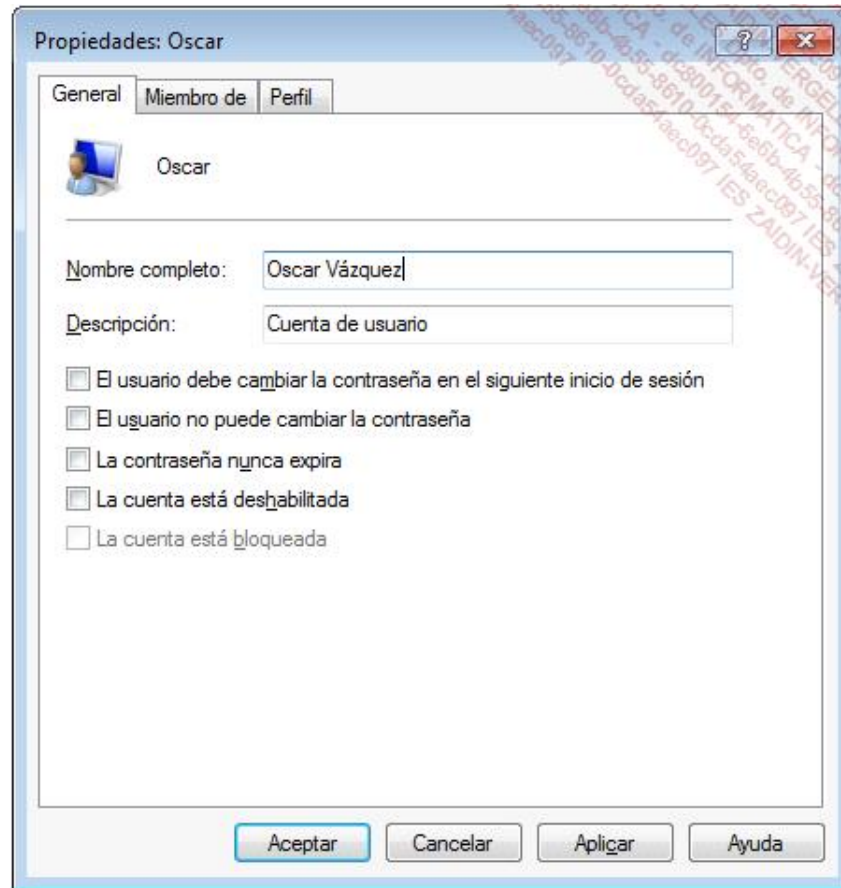
Ahora, supongamos que queremos desactivar la cuenta de Oscar ya que no se ha conectado desde hace más de 30 días. ¿Cómo podríamos hacer esto?

Pues bien, podríamos hacerlo por lo menos dos maneras diferentes: la primera consistiría en modificar la propiedad invisible `AccountDisabled` y la segunda modificar el valor de la propiedad `UserFlags`.

Primera técnica:

Modificación de una cuenta con la propiedad `AccountDisabled`.

Para conocer el estado actual de la cuenta usando la interfaz gráfica, bastará con ir al administrador del equipo, y luego en **Usuarios y grupos locales/Usuarios** y ver las propiedades del usuario. Obtendremos lo siguiente:



Verificación del estado de una cuenta con la consola de administración del equipo

Para obtener la misma información en PowerShell, vamos a ver el valor de la propiedad `AccountDisabled`:

```
PS > $user.PSBase.InvokeGet('AccountDisabled')
False
```

No devuelve el valor `False`, lo que corresponde a una cuenta activa.

■ Modifiquemos entonces este valor de la siguiente forma:

```
PS > $user.PSBase.InvokeSet('AccountDisabled', $True)
PS > $user.PSBase.CommitChanges()
```

■ Verificamos que el valor se ha modificado correctamente:

```
PS > $user=[ADSI]'WinNT://./Oscar,user'
PS > $user.PSBase.InvokeGet('AccountDisabled')
True
```



Para estar seguros de que las modificaciones han sido tenidas en cuenta por el sistema, hay que volver a cargar el objeto.

Segunda técnica:

Modificación de la propiedad `UserFlags`.

En primer lugar, miramos el valor de esta propiedad:

```
PS > $user.PSBase.InvokeGet('UserFlags')
545
```

Este valor es un «flag» o bandera al que corresponde uno o varios estados.



La lista completa de flags se encuentra aquí: [http://msdn.microsoft.com/en-us/library/aa772300\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa772300(VS.85).aspx) (`ADS_USER_FLAG_ENUM` Enumeration).

Para desactivar la cuenta, debemos efectuar una operación lógica de tipo OR con el flag `ADS_UF_ACCOUNTDISABLE = 2`, como la siguiente:

```
PS > $UserFlags=$user.PSBase.InvokeGet('UserFlags')
PS > $user.PSBase.InvokeSet('UserFlags', $($UserFlags -bor 2))
PS > $user.PSBase.CommitChanges()
```

Verificamos que el valor se ha modificado correctamente:

```
PS > $user=[ADSI]'WinNT://./Oscar,user'
PS > $user.PSBase.InvokeGet('AccountDisabled')
True
```

d. Eliminar un usuario local

La eliminación de un usuario se realiza con la ayuda del método `Delete`.

```
# Remove-LocalUser.ps1
param ([String]$maquina='.',
      [String]$Usuario=$(Throw "¡Nombre del usuario obligatorio!"))

$conexion = [ADSI]"WinNT://$maquina"
$conexion.Delete('user', $Usuario)
```

Ejemplo de utilización:

```
PS > ./Remove-LocalUser.ps1 -usuario Oscar
```