

Administración de Sistemas en la Cloud

---

# Automatización de instalación y configuración en Windows

# Índice

Esquema	3
Ideas clave	4
8.1. Introducción y objetivos	4
8.2. Instalación de Directorio Activo	5
8.3. Creación de usuarios y grupos	17
8.4. Referencias bibliográficas	23
A fondo	25
Test	26



## 8.1. Introducción y objetivos

Los comandos y sintaxis de PowerShell tiene poco valor, hasta que se ponen en un contexto más realista. Es necesario familiarizarse con ellos, pero no se aprecia su potencia hasta que no se integran en una tarea compleja.

En este tema, se presentan *scripts* de instalación de Directorio Activo. Uno de ellos configura el sistema, instala los roles y herramientas y crea el bosque y el primer dominio. El segundo creará unos pocos usuarios y grupos. Los *scripts* se analizarán paso a paso.

Los **objetivos** que se pretenden conseguir son:

- ▶ Tomar contacto con *scripts* complejos de PowerShell.
- ▶ Aprender a integrar *cmdlets* de PowerShell con otras herramientas de línea de comandos para construir *scripts* avanzados.

A continuación, en el vídeo *Paso a paso de instalación de Directorio Activo*, se demostrará, paso a paso, la ejecución de los *scripts* de este tema.



Accede al vídeo

## 8.2. Instalación de Directorio Activo

Active Directory se presentó como un servicio de directorio y de administración centralizada para equipos Windows. Una de las secciones era una guía de instalación en la que se explicaban algunos de los conceptos. El objetivo del primer *script* de este tema es automatizar esa instalación.

Se podría pensar que la creación de un bosque o de un dominio no debería automatizarse, porque es un momento muy crítico, o que no merece la pena, porque no ocurre de manera habitual. Sin embargo, también se puede argumentar a favor de esta **automatización**:

- ▶ Aunque este paso no ocurra a menudo en un entorno de producción, un *script* de este tipo permite acelerar la creación de entornos de prueba o de desarrollo, facilitando la tarea de desarrolladores y administradores.
- ▶ La automatización reduce el error humano, lo que la convierte en una herramienta ideal para un paso crítico como la creación del dominio.
- ▶ Un *script* puede incorporar controles y verificaciones que la instalación gráfica de Windows no tiene. Por ejemplo, una organización puede requerir que el nombre del dominio siga un formato concreto. También se podría integrar el *script* con una herramienta de gestión de contraseñas para almacenar la contraseña de recuperación de AD, evitando así que un administrador se olvidara o introdujera una errata al escribirla mal.

### ***Script***

A continuación, se muestra el *script* completo, listo para poder copiarlo a un fichero ps1. Los apartados siguientes comentarán cada una de las secciones.

```

<#
.SYNOPSIS
    Script de creacion de dominio
.DESCRIPTION
    Este script instalara los roles y características de Windows
    necesarias para que el servidor funcione como un controlador de
    dominio. A continuacion creara un bosque con un dominio y reiniciara
    el equipo. Tambien configurara el firewall para permitir el
    trafico relacionado con AD.
.PARAMETER Dominio
    El nombre del nuevo dominio
.PARAMETER Dns1
    La IP del servidor DNS primario
.PARAMETER Dns2
    La IP del servidor DNS secundario
.PARAMETER Password
    La contraseña que se estableciera para el usuario Administrator en el
    directorio activo (AD).
.INPUTS
    No espera datos por la entrada estandar.
.OUTPUTS
    No devuelve datos por la salida estandar.
.EXAMPLE
    PS C:/>instalar-ad.ps1 -dominio demo.loc -dns1 8.8.8.8 -dns2 8.8.4.4 -
    password secreto
.NOTES

#>

param (
    [string]$Dominio = "demo.loc",
    [string]$Dns1 = "8.8.8.8",
    [string]$Dns2 = "8.8.4.4",
    [Parameter(Mandatory=$true)][string]$Password
)

# inicializar y verificar los componentes del nombre de dominio
$componentes = $Dominio.Split(".")
If ($componentes.Length -ne 2) {

```

```

        Write-Host "El nombre de dominio debe tener el formato NOMBRE.SUFIJO"
        Exit 1
    }
    $netbios, $suffix = $componentes[0], $componentes[1]

    # guardar la contraseña como un objeto seguro
    $PassSegura = ConvertTo-SecureString -String "$Password" `
        -AsPlainText -Force

    # cambiar la contraseña del administrador
    $admin = [adsi]('WinNT://./administrator, user')
    $admin.psbase.invoke('SetPassword', $Password)

    # configurar reglas de firewall necesarias
    $reglas = ("Remote Administration",
        "File and Printer Sharing",
        "Remote Service Management",
        "Performance Logs and Alerts",
        "Remote Event Log Management",
        "Remote Volume Management",
        "Remote Scheduled Tasks Management",
        "Remote Desktop",
        "Windows Firewall Remote Management")

    ForEach ($regla in $reglas) {
        & netsh advfirewall firewall set rule group="$regla" new enable =yes
    }

    # configurar los servidores dns en la NIC
    $nicConfig = Get-WmiObject Win32_NetworkAdapterConfiguration `
        -Filter "ipenabled = 'true'" | Select-Object -First 1
    $nic = Get-WmiObject Win32_NetworkAdapter `
        -Filter "InterfaceIndex = $($nicConfig.InterfaceIndex)"
    & netsh interface ip set dnsservers name="$($nic.NetConnectionID)" `
        source=static address=$Dns1
    & netsh interface ip add dnsservers name="$($nic.NetConnectionID)" `
        address=$Dns2 index=2
    # instalar los servicios y herramientas del directorio activo
    Start-Job -Name addFeature -ScriptBlock {

```

```

Add-WindowsFeature "RSAT-AD-Tools"

$features = ("AD-Domain-Services", "DNS", "GPMC")
ForEach ($feature in $features) {
    Add-WindowsFeature -Name $feature -IncludeAllSubFeature `
        -IncludeManagementTools
}
}

Wait-Job -Name addFeature
Receive-Job -Name addFeature
if (! $?) {
    Write-Host "Ha ocurrido un error durante la instalacion de los roles"
    Exit 2
}

# crear el bosque y el dominio
Start-Job -Name addForest -ArgumentList $Dominio, $PassSegura, $netbios `
    -ScriptBlock {
        Import-Module ADDSDeployment
        Install-ADDSForest -DomainName $args[0] `
            -SafeModeAdministratorPassword $args[1] `
            -DomainNetbiosName $args[2] `
            -DomainMode Default -ForestMode Default `
            -DatabasePath "%SYSTEMROOT%\NTDS" -LogPath "%SYSTEMROOT%\NTDS" `
            -SysvolPath "%SYSTEMROOT%\SYSVOL" `
            -InstallDns -Force
    }
Wait-Job -Name addForest
Receive-Job -Name addForest
if (! $?) {
    Write-Host "Ha ocurrido un error durante la creacion del bosque"
    Exit 3
}

```



## Ayuda interactiva

El *script* arranca con el comentario con formato especial, que PowerShell interpretará como la ayuda del *script*.

```
<#
.SYNOPSIS
    Script de creacion de dominio
.DESCRIPTION
    Este script instalara los roles y características de Windows
    necesarias para que el servidor funcione como un controlador de
    dominio. A continuacion creara un bosque con un dominio y reiniciara
    el equipo. Tambien configurara el firewall para permitir el
    trafico relacionado con AD.
.PARAMETER Dominio
    El nombre del nuevo dominio
.PARAMETER Dns1
    La IP del servidor DNS primario
.PARAMETER Dns2
    La IP del servidor DNS secundario
.PARAMETER Password
    La contraseña que se estableciera para el usuario Administrator en el
    directorio activo (AD).
.INPUTS
    No espera datos por la entrada estandar.
.OUTPUTS
    No devuelve datos por la salida estandar.
.EXAMPLE
    PS C:/>instalar-ad.ps1 -dominio demo.loc -dns1 8.8.8.8 -dns2 8.8.4.4 -
    password secreto
.NOTES

#>
```

Al igual que con un *cmdlets*, `Get-Help` mostrará la ayuda al pasar el nombre del *script* como parámetro (ver Figura 1).

```

PS C:\temp> Get-Help .\install.ps1 -Detailed

NAME
    C:\temp\install.ps1

SYNOPSIS
    Script de creacion de dominio

SYNTAX
    C:\temp\install.ps1 [[-Dominio] <String>] [[-Dns1] <String>] [[-Dns2] <String>] [-Password] <String>
    [<CommonParameters>]

DESCRIPTION
    Este script instalara los roles y caracteristicas de Windows
    necesarias para que el servidor funcione como un controlador de
    dominio. A continuacion creara un bosque con un dominio y reiniciara
    el equipo. Tambien configurara el firewall para permitir el
    trafico relacionado con AD.

PARAMETERS
    -Dominio <String>
        El nombre del nuevo dominio

    -Dns1 <String>
        La IP del servidor DNS primario

    -Dns2 <String>
        La IP del servidor DNS secundario

    -Password <String>
        La contraseña que se establecera para el usuario Administrator en el directorio activo (AD).

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (https://go.microsoft.com/fwlink/?LinkID=113216).

----- EXAMPLE 1 -----
PS C:/>instalar-ad.ps1 -dominio demo.loc -dns1 8.8.8.8 -dns2 8.8.4.4 -password secreto

```

Figura 1. Ayuda integrada del *script*. Fuente: elaboración propia.

## Parámetros

PowerShell ofrece una funcionalidad integrada para el «parseo» de parámetros. Este *script* solo necesita parámetros de tipo cadena, pero PowerShell se encargaría de verificar el tipo si alguno de ellos fuera de tipo entero.

```

param (
    [string]$Dominio = "demo.loc",
    [string]$Dns1 = "8.8.8.8",
    [string]$Dns2 = "8.8.4.4",
    [Parameter(Mandatory=$true)][string]$Password
)

```

La verificación de tipo no tiene por qué validar los valores completamente. Por ejemplo, los nombres de dominio deben seguir el esquema de nombres DNS. Se podría obviar la comprobación de este formato y dejar que la creación del dominio

falle más adelante. Sin embargo, se puede mejorar la experiencia del usuario si el *script* detecta el problema al principio.

En este caso concreto, el formato se puede verificar dividiendo el parámetro `$dominio` entre los puntos de la cadena de texto, de manera que, si el `$dominio` es "demo.loc", `$componentes` es una lista con dos cadenas, "demo" y "loc". El nombre del dominio no tiene por qué estar limitado a dos componentes, pero una organización podría establecer este requisito como una política interna.

```
# inicializar y verificar los componentes del nombre de dominio
$componentes = $Dominio.Split(".")
If ($componentes.Length -ne 2) {
    Write-Host "El nombre de dominio debe tener el formato NOMBRE.SUFIJO"
    exit 3
}
```

Las siguientes líneas manipulan los parámetros para construir variables necesarias más adelante. El nombre del dominio completo es necesario tal cual, pero, en algunos casos, es necesario usar los componentes por separado. Además, las contraseñas deben almacenarse como un objeto `SecureString`.

```
$netbios, $suffix = $componentes[0], $componentes[1]

# guardar la contraseña como un objeto seguro
$PassSegura = ConvertTo-SecureString -String "$Password" `
    -AsPlainText -Force
```

Este *script* intenta ser 100 % automatizable, pero en una ejecución interactiva se podría haber optado por leer la contraseña por la línea de comandos (ver Figura 2). Este paso requiere una acción del usuario, por lo que no es ideal para un *script* de automatización.

```
PS C:\temp> $SecureInput = Read-Host -AsSecureString
*****
PS C:\temp> $SecureInput
System.Security.SecureString
PS C:\temp>
```

Figura 2. Contraseña como cadena segura. Fuente: elaboración propia.

## Cambio de la contraseña local

Este paso modifica la contraseña del administrador al valor introducido como parámetro. La cuenta de administrador local se convertirá en administrador del dominio, por lo que este paso es parte de la configuración de AD. No es estrictamente necesario, pero se incluye como ejemplo de una las integraciones de PowerShell.

PowerShell no solo permite el uso de *cmdlets* o de binarios. Este ejemplo hace uso de una integración con un objeto [COM](#), una interfaz de intercambio de objetos entre procesos propia de Microsoft. Aunque es muy antigua, sigue estando disponible y, aunque suele haber comandos de PowerShell equivalentes, el uso de esta interfaz permite que las organizaciones reutilicen *scripts* antiguos sin necesidad de reescribirlos como comandos nativos de PowerShell.

La interfaz COM del ejemplo es [ADSI](#). Aunque está pensada para acceder a objetos de AD, en este caso se conecta al equipo local (en este punto del *script* aún no se ha creado el dominio). La cadena "WinNT://./administrator" indica una conexión al objeto administrator del equipo local, identificado por un «.».

```
# cambiar la contraseña del administrador
$admin = [adsisearcher]'WinNT://./administrator, user'
$admin.psbase.invoke('SetPassword', $Password)
```

## Configuración del *firewall*

Al igual que la contraseña del administrador, la configuración del *firewall* de Windows no es estrictamente una parte necesaria de la creación del dominio. Sirve, no obstante, para mostrar dos funcionalidades como ejemplo.

Por un lado, el *script* hace uso de un bucle *foreach* para ejecutar el mismo comando tantas veces como reglas hay que configurar. De esa manera, si hay que cambiar el comando, solo es necesario editar una línea.

La segunda funcionalidad es la ejecución de comandos arbitrarios desde PowerShell. El *script* usa [netsh](#), una utilidad de configuración de interfaces de red para Windows. PowerShell dispone de varias opciones para arrancar programas externos. Una de ellas es Start-Process. La que usa el *script* es el símbolo «&», que ejecuta el resto de la línea en una consola Windows, o cmd.exe, no en una consola de PowerShell.

PowerShell dispone de *cmdlets* para configurar el *firewall*, tales como Set-NetFirewallRule y New-NetFirewallRule, que podrían usarse nativamente en el *script*. Este bloque debe considerarse únicamente como un ejemplo.

```
# configurar reglas de firewall necesarias
$reglas = ("Remote Administration",
           "File and Printer Sharing",
           "Remote Service Management",
           "Performance Logs and Alerts",
           "Remote Event Log Management",
           "Remote Volume Management",
           "Remote Scheduled Tasks Management",
           "Remote Desktop",
           "Windows Firewall Remote Management")

ForEach ($regla in $reglas) {
    & netsh advfirewall firewall set rule group="$regla" new enable =yes
}
```

## Servidores DNS

El siguiente bloque configura los servidores DNS como valores estáticos, la configuración recomendada para AD. El *script* vuelve a hacer uso de netsh para la configuración, pero usa la interfaz [WMI](#) para recuperar el ID de la tarjeta de red. WMI es una interfaz para la administración de múltiples objetos del sistema operativo. No es una interfaz exclusiva de PowerShell; de hecho, era un método muy común en VBScript, antes de que se extendiera el uso de PowerShell.

```
# configurar los servidores dns en la NIC
$nicConfig = Get-WmiObject Win32_NetworkAdapterConfiguration `
    -Filter "ipenabled = 'true'" | Select-Object -First 1
$nic = Get-WmiObject Win32_NetworkAdapter `
    -Filter "InterfaceIndex = $($nicConfig.InterfaceIndex)"
& netsh interface ip set dnsservers name="$($nic.NetConnectionID)" `
    source=static address=$Dns1
& netsh interface ip add dnsservers name="$($nic.NetConnectionID)" `
    address=$Dns2 index=2
```

## Instalación de roles y características

En este punto del *script*, la configuración preliminar ya está lista. El siguiente paso consiste en la instalación de los servicios y herramientas de administración necesarios. Las llamadas al *cmdlet* `Add-WindowsFeature` se podrían usar directamente, ya que son síncronas (es decir, bloquean la consola sin poder progresar hasta que terminan), pero el *script* hace uso de `Start-Job` para demostrar el uso de trabajos **de fondo** (o de *background*).

El *cmdlet* `Start-Job` arranca un proceso de PowerShell adicional, independiente del proceso que ejecuta el *script*. Ese segundo proceso ejecuta el código del bloque indicado por el parámetro `ScriptBlock`. No tiene acceso al ámbito del proceso principal, por lo que no puede usar las variables definidas en el *script* principal. A todos los efectos, es igual que arrancar un programa y dejar que se ejecute.

Una vez arrancado, los *cmdlets* `Wait-Job` y `Receive-Job` se encargan de esperar a que el trabajo termine y de recibir la salida del trabajo, si la hubiera, respectivamente. Si el trabajo ha terminado satisfactoriamente, el *script* principal continuará su ejecución. Por el contrario, si el trabajo terminó con error, la variable automática «\$?» contendrá `False` y el *script* terminará con un mensaje al usuario y un código de error específico, `Exit 2`. La variable «\$?» es la misma que en Bash contiene el código de salida, pero, mientras que en Bash es un entero, en PowerShell es un booleano.

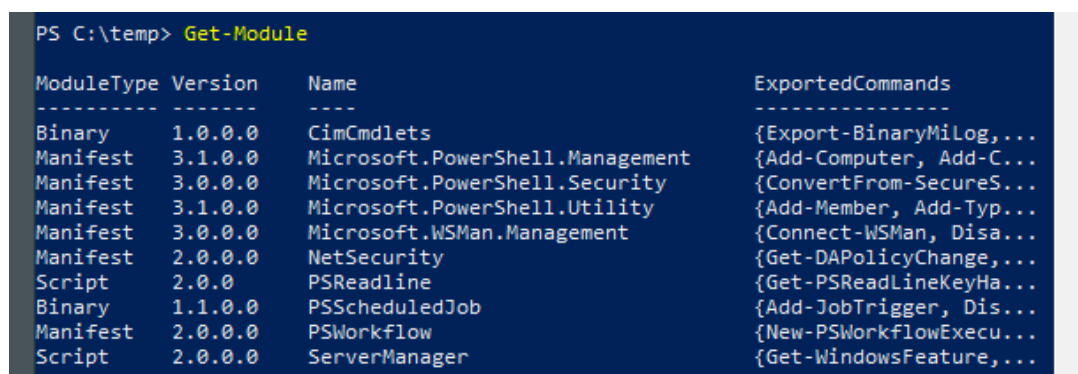
```
# instalar los servicios y herramientas del directorio activo
Start-Job -Name addFeature -ScriptBlock {
    Add-WindowsFeature "RSAT-AD-Tools"

    $features = ("AD-Domain-Services", "DNS", "GPMC")
    ForEach ($feature in $features) {
        Add-WindowsFeature -Name $feature -IncludeAllSubFeature `
            -IncludeManagementTools
    }
}
Wait-Job -Name addFeature
Receive-Job -Name addFeature
if (! $?) {
    Write-Host "Ha ocurrido un error durante la instalacion de los roles"
    Exit 2
}
```

## Creación del bosque y del primer dominio

El último bloque sigue el mismo esquema que el anterior al crear un trabajo para ejecutar un bloque de código. Sin embargo, en este caso, el bloque de código necesita acceder a las variables del *script* principal. Para ello, el *cmdlet* `Start-Job` acepta una lista de parámetros que el bloque recibirá en la variable `$args`. Estos argumentos ya no tienen nombre, por lo que hay que acceder a ellos con el orden en el que se pasan a `ArgumentList`.

Este paso, además, muestra el uso de `Import-Module`. PowerShell es extensible mediante módulos, pero estos módulos no tienen por qué estar cargados por defecto en cualquier consola. La Figura 3 muestra los módulos importados por defecto en una sesión (esto puede variar de un equipo a otro, en función de los roles instalados). Una vez importado un módulo, los *cmdlets* definidos por este se pueden usar en el resto de la sesión.



ModuleType	Version	Name	ExportedCommands
Binary	1.0.0.0	CimCmdlets	{Export-BinaryMilog,...}
Manifest	3.1.0.0	Microsoft.PowerShell.Management	{Add-Computer, Add-C...}
Manifest	3.0.0.0	Microsoft.PowerShell.Security	{ConvertFrom-SecureS...}
Manifest	3.1.0.0	Microsoft.PowerShell.Utility	{Add-Member, Add-Typ...}
Manifest	3.0.0.0	Microsoft.WSMan.Management	{Connect-WSMan, Disa...}
Manifest	2.0.0.0	NetSecurity	{Get-DAPolicyChange,...}
Script	2.0.0	PSReadline	{Get-PSReadLineKeyHa...}
Binary	1.1.0.0	PSScheduledJob	{Add-JobTrigger, Dis...}
Manifest	2.0.0.0	PSWorkflow	{New-PSWorkflowExecu...}
Script	2.0.0.0	ServerManager	{Get-WindowsFeature,...}

Figura 3. Módulos importados por defecto. Fuente: elaboración propia.

El cmdlet `Install-ADDSForest` es similar y acepta los mismos parámetros.

```
# crear el bosque y el dominio
Start-Job -Name addForest -ArgumentList $Dominio, $PassSegura, $netbios `
-ScriptBlock {
    Import-Module ADDSDeployment
    Install-ADDSForest -DomainName $args[0] `
        -SafeModeAdministratorPassword $args[1] `
        -DomainNetbiosName $args[2] `
        -DomainMode Default -ForestMode Default `
        -DatabasePath "%SYSTEMROOT%\NTDS" -LogPath "%SYSTEMROOT%\NTDS" `
        -SysvolPath "%SYSTEMROOT%\SYSVOL" `
        -InstallDns -Force
}
Wait-Job -Name addForest
Receive-Job -Name addForest
if (! $?) {
    Write-Host "Ha ocurrido un error durante la creacion del bosque"
    Exit 3
}
```



Una vez completada la instalación, el equipo se reiniciará y ya será necesario iniciar sesión con la cuenta de administrador de dominio.

## 8.3. Creación de usuarios y grupos

El siguiente *script* sirve de ejemplo de creación de recursos de Directorio Activo. Se ha separado del *script* anterior, ya que solo tiene sentido crear recursos una vez que existe el dominio.

```
<#
.SYNOPSIS
    Script de creacion de usuarios y grupos de AD
.DESCRIPTION
    Este script instalara inicializa un dominio de AD con varios usuarios
    de prueba, una OU nueva para grupos, y varios grupos en esa OU a
    los que agregara los usuarios como miembros. El script espera dos
    contraseñas: la del administrador de dominio para crear la OU
    y la contraseña por defecto de los usuarios.
.PARAMETER Dominio
    El nombre del dominio
.PARAMETER PasswordAdmin
    La contraseña del Administrador del dominio.
.PARAMETER PasswordUsuarios
    La contraseña por defecto de los nuevos usuarios.
.EXAMPLE
    PS C:/>crear-grupos.ps1 -Dominio demo.loc -PasswordAdmin secreto1 -
    PasswordUsuarios secreto2
.NOTES

#>

param (
    [string]$Dominio = "demo.loc",
    [Parameter(Mandatory=$true)][string]$PasswordAdmin,
    [Parameter(Mandatory=$true)][string]$PasswordUsuarios
```

```

)

# inicializar y verificar los componentes del nombre de dominio
$componentes = $Dominio.Split(".")
If ($componentes.Length -gt 2) {
    Write-Host "El nombre de dominio debe tener el formato NOMBRE.SUFIJO"
    exit 3
}
$netbios, $suffix = $componentes[0], $componentes[1]

# guardar la contraseña como un objeto seguro
$PassSeguraAdmin = ConvertTo-SecureString -String "$PasswordAdmin" `
    -AsPlainText -Force

$PassSeguraUser = ConvertTo-SecureString -String "$PasswordUsuarios" `
    -AsPlainText -Force

# crear la unidad organizativa
$credencial = New-Object
System.Management.Automation.PSCredential("$netbios\Administrator",
$PassSeguraAdmin)
New-ADOrganizationalUnit -Credential $credencial -Name Groups `
    -Path "DC=$netbios,DC=$suffix"

# crear 5 usuarios de prueba
$a = 1
Do
{
    "Creando el usuario de pruebas: Tester_{$a}"
    New-ADUser -Credential $credencial -Name Tester-{$a} `
        -Path "CN=Users,DC=$netbios,DC=$suffix" `
        -SamAccountName admin-tester-{$a} `
        -GivenName Tester-{$a} -SurName Test-Tested-{$a} `
        -AccountPassword $PassSeguraUser -Description Testing-User-{$a} `
        -UserPrincipalName admin-tester-{$a}@$Dominio `
        -DisplayName Tester-Test-Tested-{$a} `
        -PasswordNeverExpires $false -Enabled $true
    $a++
} While ($a -le 5)

```

```
# crear 3 grupos y añadir todos los usuarios a todos los grupos
$g = 1
Do
{
    New-ADGroup -Credential $credencial -Name "Administrators_$g" `
        -GroupScope Global `
        -Path "OU=Groups,DC=$netbios,DC=$suffix" 2>&1 | out-null
    $a = 1
    Do
    {
        "Agregando el usuario Tester_$a al grupo Administrators_$g"
        Add-ADGroupMember -Credential $credencial `
            -identity "CN=Administrators_$g,OU=Groups,DC=$netbios,DC=$suffix" `
            -Members "CN=Tester-$a,CN=Users,DC=$netbios,DC=$suffix" 2>&1 | out-
null
        $a++
    } While ($a -le 5)
    $g++
} While ($g -le 3)
```

## Inicialización y parámetros

Estos bloques son muy parecidos a los del apartado anterior, por lo que no se entrará en detalle. El único apartado relevante es que el *script* **acepta dos contraseñas**: la contraseña del administrador que ejecuta el *script* y la contraseña por defecto que tendrán los usuarios.

En un caso real, sería habitual que el *script* de creación de cuentas generara una contraseña temporal aleatoria que se hiciera llegar al usuario por algún mecanismo (un *email*, si el usuario dispone de buzón, o un *email* a su responsable, por ejemplo). En ese caso, el administrador no llegaría nunca a conocer la primera contraseña del usuario.

## Creación de OU

El primer paso consiste en crear la unidad organizativa que contendrá las cuentas del grupo. El *cmdlet* `New-ADOrganizationalUnit` requiere un tipo de objeto `PSCredential` para la contraseña, diferente a los del anterior *script*.

```
# crear la unidad organizativa
$credencial = New-Object
System.Management.Automation.PSCredential("$netbios\Administrator",
$PassSeguraAdmin)
New-ADOrganizationalUnit -Credential $credencial -Name Groups `
-Path "DC=$netbios,DC=$suffix"
```

## Creación de usuarios

El ejemplo de creación de usuarios no tiene por qué ser el más útil en un entorno real, pero puede ser habitual en entornos de prueba, donde hace falta generar varias cuentas de usuario de manera rápida. El bloque del bucle, un *do-while*, inicializa muchos parámetros de la cuenta del usuario a partir de una variable índice.

```
# crear 5 usuarios de prueba
$a = 1
Do
{
    "Creando el usuario de pruebas: Tester_$a"
    New-ADUser -Credential $credencial -Name Tester-$a `
        -Path "CN=Users,DC=$netbios,DC=$suffix" `
        -SamAccountName admin-tester-$a `
        -GivenName Tester-$a -SurName Test-Tested-$a `
        -AccountPassword $PassSeguraUser -Description Testing-User-$a `
        -UserPrincipalName admin-tester-$a@$Dominio `
        -DisplayName Tester-Test-Tested-$a `
        -PasswordNeverExpires $false -Enabled $true
    $a++
} While ($a -le 5)
```

## Creación de grupos

El último bloque sigue un esquema similar al anterior, pero con dos bucles anidados: el externo crea los grupos y el interno añade todos los usuarios como miembros del grupo.

La salida de los comandos `New-ADGroup` y `Add-ADGroupMember` envía tanto la salida estándar como la salida de errores a `Out-Null`. Este *cmdlet* es similar al dispositivo `/dev/null` de Linux: borra los datos, en vez de redirigirlos por la tubería. El objetivo es evitar que aparezca un error si ya existe un grupo o si un miembro ya ha sido añadido.

```
# crear 3 grupos y añadir todos los usuarios a todos los grupos
$g = 1
Do
{
    New-ADGroup -Credential $credencial -Name "Administrators_$g" `
        -GroupScope Global `
        -Path "OU=Groups,DC=$netbios,DC=$suffix" 2>&1 | out-null
    $a = 1
    Do
    {
        "Agregando el usuario Tester_$a al grupo Administrators_$g"
        Add-ADGroupMember -Credential $credencial `
            -identity "CN=Administrators_$g,OU=Groups,DC=$netbios,DC=$suffix" `
            -Members "CN=Tester-$a,CN=Users,DC=$netbios,DC=$suffix" 2>&1 | out-
null
        $a++
    } While ($a -le 5)
    $g++
} While ($g -le 3)
```

Además, tanto este bloque como el anterior incluyen cadenas de texto sin un *cmdlet* a la vista. El comportamiento de PowerShell es escribir estos datos por la salida estándar, de manera similar a volcarlos con `Write-Host`. El resultado es igual a una línea de *log* (ver Figura 4).

```

PS C:\temp> .\create.ps1 -Dominio demo.loc -PasswordAdmin Password1! -PasswordUsuarios Prueba1234!
Creando el usuario de pruebas: Tester_1
Creando el usuario de pruebas: Tester_2
Creando el usuario de pruebas: Tester_3
Creando el usuario de pruebas: Tester_4
Creando el usuario de pruebas: Tester_5
Agregando el usuario Tester_1 al grupo Administrators_1
Agregando el usuario Tester_2 al grupo Administrators_1
Agregando el usuario Tester_3 al grupo Administrators_1
Agregando el usuario Tester_4 al grupo Administrators_1
Agregando el usuario Tester_5 al grupo Administrators_1
Agregando el usuario Tester_1 al grupo Administrators_2
Agregando el usuario Tester_2 al grupo Administrators_2
Agregando el usuario Tester_3 al grupo Administrators_2
Agregando el usuario Tester_4 al grupo Administrators_2
Agregando el usuario Tester_5 al grupo Administrators_2
Agregando el usuario Tester_1 al grupo Administrators_3
Agregando el usuario Tester_2 al grupo Administrators_3
Agregando el usuario Tester_3 al grupo Administrators_3
Agregando el usuario Tester_4 al grupo Administrators_3
Agregando el usuario Tester_5 al grupo Administrators_3
PS C:\temp>

```

Figura 4. Ejecución del *script* de creación de usuarios y grupos. Fuente: elaboración propia.

## Comprobación

El hecho de poder iniciar sesión en el equipo tras el reinicio es prueba suficiente de que el primer *script* funcionó como se esperaba y, si el segundo *script* termina sin errores, es muy probable que las cuentas de usuario ya estén disponibles. Se podría comprobar en la consola gráfica, abriendo el panel de Active Directory Users and Computers desde el Server Manager, tal como muestra la Figura 5.

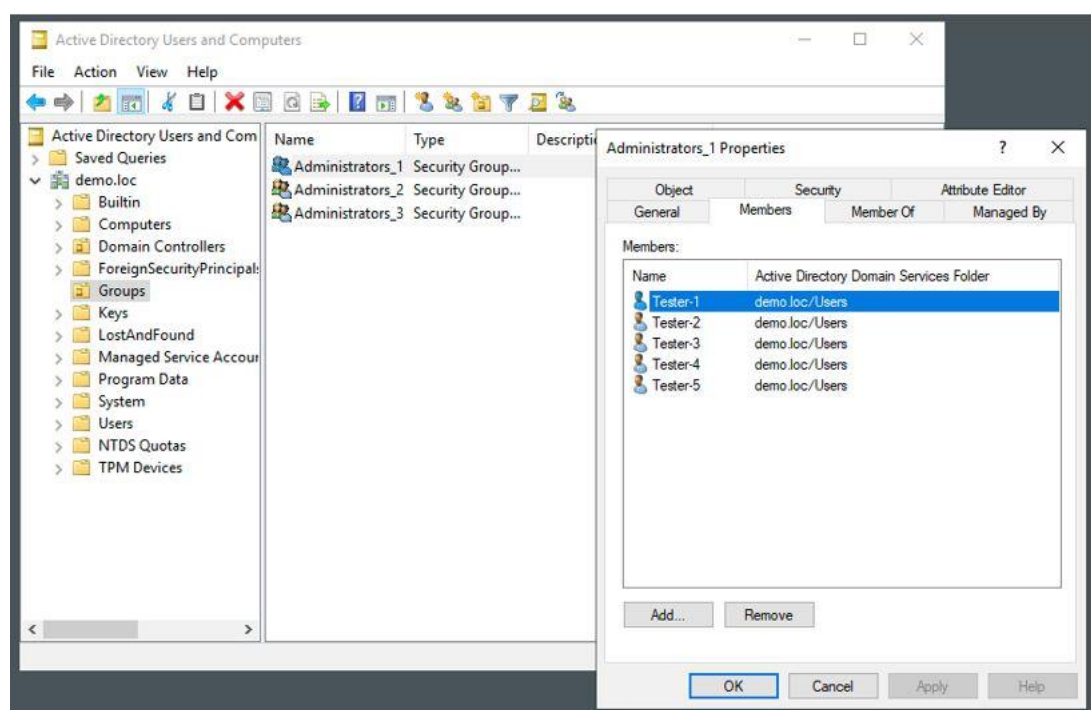


Figura 5. Consola de AD con los usuarios recién creados. Fuente: elaboración propia.

También es posible, dado que el tema trata sobre PowerShell, consultar el Directorio Activo directamente en la línea de comandos:

```
PS C:\> $ou = (Get-ADOrganizationalUnit -Filter 'Name -Like "Groups"')
PS C:\> $ou.Name
Groups

PS C:\> Get-ADGroup -Filter * -SearchBase $ou.DistinguishedName | Select-Object -Property SamAccountName,SID

SamAccountName  SID
-----
Administrators_1 S-1-5-21-3436996036-797900140-2994447757-1126
Administrators_2 S-1-5-21-3436996036-797900140-2994447757-1127
Administrators_3 S-1-5-21-3436996036-797900140-2994447757-1128
```

## 8.4. Referencias bibliográficas

Microsoft. (2018, mayo 31). *ADSI Interfaces*. <https://docs.microsoft.com/en-us/windows/win32/adsi/adsi-interfaces>

Microsoft. (2018, mayo 31). *Component Object Model (COM)*. <https://docs.microsoft.com/en-gb/windows/win32/com/component-object-model-com--portal?redirectedfrom=MSDN>

Microsoft. (2018, mayo 31). *Windows Management Instrumentation*. <https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page>

Microsoft. (2020, julio 8). *Netsh Command Syntax, Contexts, and Formatting*. <https://docs.microsoft.com/en-us/windows-server/networking/technologies/netsh/netsh-contexts>

Shepard, M. (2015). *Getting Started with PowerShell*. Packt Publishing.



## PowerShell en Windows Server 2019

Krause, J. (2019). *Mastering Windows Server 2019: The Complete Guide for IT Professionals to Install and Manage Windows Server 2019 and Deploy New Capabilities*. (5.ª ed.). Packt Publishing.

El capítulo de PowerShell de este libro ofrece muchos ejemplos similares a los de los *scripts* de este tema, como, por ejemplo, cómo unir un ordenador al dominio o cómo configurar las direcciones IP de una tarjeta de red. Es un buen complemento para ampliar los conocimientos sobre PowerShell.

1. ¿Qué contiene «\$?» cuando el último proceso ha terminado con éxito?
  - A. 0.
  - B. -1.
  - C. Null.
  - D. \$true.
  
2. ¿Cómo se activan los *cmdlets* de un módulo en una sesión de PowerShell?
  - A. Import-Module.
  - B. Import-Package.
  - C. Get-Module.
  - D. Install-Module.
  
3. ¿Cómo se indica que un parámetro es obligatorio en un *script*?
  - A. -Force.
  - B. [Parameter(Mandatory=\$true)].
  - C. [Parameter(Required=\$true)].
  - D. [mandatory][string].
  
4. ¿Por qué un bloque de *script* de un trabajo no puede acceder a las variables de la sesión actual?
  - A. Sí que puede.
  - B. Porque se reinicia el ámbito manualmente.
  - C. Porque se ejecuta en un proceso diferente.
  - D. Ninguna de las anteriores.

5. ¿Qué comando de PowerShell equivale a `/dev/null` en Linux?
- A. `Out-Null`.
  - B. `Dev-Null`.
  - C. `Write-Null`.
  - D. `/dev/null`.
6. ¿Cómo se divide una cadena `$s` de texto en subcadenas separadas por un carácter de punto y coma?
- A. `$s.Divide()`.
  - B. `Get-Split($s, ";")`.
  - C. `$s.Split(";")`.
  - D. `$s.Split()`.
7. ¿Cómo se accede a la ayuda personalizada de un *script*?
- A. `Get-Help ./nombre-script.ps1`.
  - B. `Get-Help nombre-script`.
  - C. `nombre-script /h`.
  - D. `nombre-script.ps1 --help`.
8. ¿Cómo se puede leer una cadena de texto como una contraseña segura?
- A. `Get-Password`.
  - B. `Read-Host`.
  - C. `Read-Host -AsSecureString`.
  - D. `Read-Password -Secure`.
9. ¿Qué uso de le da a esta sintaxis `[string]$PasswordAdmin`?
- A. Indica el tipo del parámetro `$PasswordAdmin` como de tipo `string`.
  - B. Permite leer una clave con permisos de administración.
  - C. Verifica que una clave sea de tipo cadena.
  - D. Todas las anteriores.

10. ¿Qué uso de le da al comando New-ADUser?

- A. Crea un nuevo AD.
- B. Permite añadir nuevos usuarios.
- C. No es un comando valido.
- D. A y B son correctas.