PowerShell Tips and Tricks for Scripting in Active Directory Test Environments

blog.netwrix.com/2023/02/10/powershell-tips-and-tricks

Joe Dibley

PowerShell is one of the most efficient management methods in the Windows Server world. This article offers tips and tricks to learn about one of the most common scripting scenarios: using PowerShell in test, demo and quality assurance (QA) environments, which frequently need to be rebuilt or adjusted to fit a new need or process.

Handpicked related content:

[Free Download] PowerShell Scripting Tutorial

We've chosen the most useful PowerShell tips based on real-world experience with colleagues and customers. We will not cover creating or configuring a new lab environment or managing a production Microsoft Active Directory environment.

Tip #1: Simplify Cmdlet Parameters with Splatting and Custom PowerShell Objects

One of the most helpful PowerShell tricks concerns specifying input parameters to PowerShell cmdlets.

We usually don't need to specify more than a handful of parameters per command in our scripting. For example, when running the New-ADUser cmdlet to create a new Active Directory user, we might specify just the user's first name, last name and display name, as follows:

```
New-ADUser -FirstName "Alex" -LastName "Smith" -DisplayName "Alex Smith"
```

However, the New-ADUser command actually has over 63 possible parameters! If you need to use a large number of them, your script can quickly become hard to manage and troubleshoot.

Thankfully, there are two excellent ways to efficiently specify parameters when using PowerShell cmdlets:

Using Splatting

Splatting is a method of passing a collection of parameters and values to a command as a unit. For instance, let's create a hashtable called UserObject that contains all the parameters and their values for a new user we want to create:

```
$UserObject = @{
    firstName= "Alex"
    lastName = "Smith"
    displayName    = "Alex Smith"
    title = "The Boss"
    department = "The Boss Department"
}
```

Now we can splat this hashtable to the New-ADUser cmdlet by specifying UserObject with @ instead of $:

```
New-AdUser @UserObject
```

When the command is run, PowerShell interprets it as:

```
New-AdUser @UserObject -firstName “Alex” -lastName “Smith” -displayName “Alex
Smith” -title “The Boss” -department “The Boss Department”
```

Now imagine having 50 parameters instead of just five. You can see how splatting enables us to make scripts more organized and adaptable!

Using Custom Objects

An alternative method for achieving the same result is to create a custom PowerShell object using a hashtable and pass that object to the cmdlet over a PowerShell pipeline.

Here’s how we can create a custom PowerShell object based on the same hashtable we used above in our splatting example:

```
$UserObject = [PSCustomObject]@{
    firstName= "Alex"
    lastName = "Smith"
    displayName    = "Alex Smith"
    title = "The Boss"
    department = "The Boss Department"
}
```

Then we would simply use a pipeline to pass the PowerShell object into the New-AdUser command:

```
$UserObject | New-AdUser
```

Note that some AD cmdlets might not accept objects from the pipeline, so make sure to check the documentation on the cmdlet you are using first.

Tip #2: Consider Alternatives to Using External Data Artifacts

PowerShell includes the following very useful cmdlets:

- **import-CSV** — Imports a CSV file to facilitate taking actions in bulk
- **export-CSV** — Exports script results to a CSV file, which can speed report creation

For example, we might create a script that creates new Active Directory users by importing a CSV file listing each user's first name and last name.

Although CSV files are undeniably useful, they can limit the portability of your scripts. If even if a script writer provides extensive comments, ensuring that external data artifacts are properly formatted can be complicated and error-prone.

Luckily, there are some ways to work around this with PowerShell.

Alternative A: Internalize the Data

While not necessarily a good practice for production script, hardcoding or semi-dynamically including data in a script is sometimes a good idea. For example, suppose we need to create a set of AD users for a demo or training session. We can use arrays, objects or hashtables to hardcode a list of names into our PowerShell script, eliminating the need for CSV files listing these details.

Here is how we'd create a set of string arrays to populate a list of first names and last names:

```
$FirstNames =
@("Alan","Arnold","Alexander","Anne","Anika","Bert","Betsy","Carl","Cynthia","David

$LastNames = @("Andel","Apel","Baros","Cernik","Danek","Filipek","Franel")
```

This is a simple example, but it is possible to easily embed hundreds of values into a single array.

We can even retrieve random values from an array using the Get-Random command:

```
$FirstNames | Get-Random
```

Alternative B: Retrieve Common Variables from Online Sources

Another alternative to using external data files is to leverage publicly available datasets, services and application programming interfaces (APIs) to dynamically populate scripts with data.

IMPORTANT: Ensure you review the content of the files you find before using them, and definitely do not use this strategy in your production environment. This is really only safe for demo environments!

One excellent resource is the SecLists GitHub repository, a collection of usernames, passwords, web shells, URLs, sensitive data patterns, and other information used in security assessments. By cloning this repository or merely using a command like Invoke-WebRequest, we can retrieve common first names, last names, locations, job titles and other items for use in our scripts.

```
# Get Lists of Names and Titles From GitHub and Public Data Sets
$FirstNames = (Invoke-WebRequest -UseBasicParsing -Method Get -Uri
'https://raw.githubusercontent.com/danielmiessler/SecLists/master/Usernames/Names/n
 | Select-Object -ExpandProperty 'Content') -split '`n'

$LastNames = (Invoke-WebRequest -UseBasicParsing -Method Get -Uri
'https://raw.githubusercontent.com/danielmiessler/SecLists/master/Usernames/Names/f
usa-top1000.txt' | Select-Object -ExpandProperty 'Content') -split '`n'

$JobTitles = (Invoke-WebRequest -UseBasicParsing -Method Get -Uri
'https://raw.githubusercontent.com/Stutern/job-titles/master/job-titles.csv' |
Select-Object -ExpandProperty 'Content') -split '`n'

$Cities = ("Denver","Seattle","Los Angeles","San Diego","Las
Vegas","Denver","Austin","Chicago","Madison","Minneapolis","Milwaukee","New
York","Newark","Houston","Prague","Berlin","Basel","Frankfurt")

$Departments = ("Accounting","Sales","Human Resources","Information
Technology","Executive","Consulting","Marketing","Retail")
```

How Netwrix Can Help

As you can see, creating scripts to help manage and secure Active Directory can be challenging, especially when you have limited time and resources.

Netwrix Auditor for Active Directory can help. It delivers complete visibility into what's going on in your Active Directory and Group Policy, and simplifies the work of auditing Active Directory logons and changes to reduce the risk of privilege abuse, streamline troubleshooting and prove IT compliance.

Joe Dibley
Security Researcher at Netwrix and member of the Netwrix Security Research Team. Joe is an expert in Active Directory, Windows, and a wide variety of enterprise software platforms and technologies, Joe researches new security risks, complex attack techniques, and associated mitigations and detections.