CloudCommand

Cyber Security

CloudCommand GmbH chr.schumacher@gmx.tm

CloudCommand GmbH chr.schumacher@gmx.tm

AGENDA

01 Fortgeschrittenes Skripting

02 Automatisierung und Scheduling

03 Verwaltung von Systemen und Netzwerken

04 Sicherheit und Best Practices

01 Fortgeschrit-tenes Skripting

Fortgeschrittenes Skripting

Themen

1. Verwendung von Funktionen und Modulen
2. Fehlerbehandlung und Debugging
3. Arbeiten mit Skript-Parameter und Argumenten

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Was ist eine Funktion?

Eine Funktion ist ein Block von Code, der eine bestimmte Aufgabe ausführt. Die Funktion ermöglicht es, Code zu organisieren und wiederzuverwenden, ohne ihn mehrfach schreiben zu müssen.

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Wie erstellt man eine Funktion?

Hier ist ein einfaches Beispiel:

```
function Say-Hello {
   param (
        [string]$Name
   )
   "Hallo, $Name!"
}
```

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Erklärung:

-

`function Say-Hello`

: Hier definieren wir eine Funktion mit dem Namen

`Say-Hello`

.

-

`param`

: Damit können wir Parameter an die Funktion übergeben. In diesem Fall erwartet die Funktion einen Namen.

- Der Code innerhalb der geschweiften Klammern führt die Aufgabe aus, in diesem Fall gibt er eine Begrüßung aus.

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Aufruf der Funktion:

```
Say-Hello -Name Klaus
```

Erklärung

```
Say-Hello # ist unser Funktion, welche wir erstellt haben
-Name     # ist unser Parameter
Klaus     # Der gewünschte Wert
```

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Was ist ein Modul?

Ein Modul ist eine Sammlung von Funktionen, Cmdlets, Variablen und anderen Ressourcen, die zusammengefasst sind. Module helfen dabei, den Code zu organisieren und wiederverwendbar zu machen.

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Wie erstellt man ein Modul?

1. Erstellen Sie eine neue Datei mit der Endung `.psm1`, z.B. `MeinModul.psm1`.

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

2. Fügen Sie Ihre Funktionen in diese Datei ein:

```
function Say-Hello {
   param (
        [string]$Name
   )
   "Hallo, $Name!"
}

function Say-Goodbye {
   param (
        [string]$Name
   )
   "Auf Wiedersehen, $Name!"
}
```

3. Speichern Sie die Datei.

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Wie lädt man ein Modul?

Um das Modul in PowerShell zu verwenden, müssen Sie es zuerst importieren:

```
Import-Module "Pfad\Zu\MeinModul.psm1"
```

Wie verwendet man die Funktionen aus dem Modul?

Nach dem Import können Sie die Funktionen wie gewohnt verwenden:

```
Say-Hello -Name "Max"
Say-Goodbye -Name "Max"
```

Fortgeschrittenes Skripting

Verwendung von Funktionen und Modulen

Zusammenfassung

- Funktionen helfen Ihnen, wiederverwendbaren Code zu erstellen, der spezifische Aufgaben ausführt.

- Module sind Sammlungen von Funktionen und anderen Ressourcen, die Ihnen helfen, Ihren Code zu organisieren und zu strukturieren.

Fortgeschrittenes Skripting

Fehlerbehandlung und Debugging

Fehlerbehandlung und Debugging in PowerShell sind wichtige Aspekte, um sicherzustellen, dass deine Skripte reibungslos laufen und Probleme schnell identifiziert werden können.

Fehlerbehandlung bedeutet, Mechanismen zu verwendest, um mit Fehlern umzugehen, die während der Ausführung deines Skripts auftreten können.

Fortgeschrittenes Skripting

Fehlerbehandlung und Debugging

1. Try-Catch-Blöcke: Du kannst einen Codeblock in `try` setzen, und wenn ein Fehler auftritt, wird der Code im `catch`-Block ausgeführt. So kannst du spezifische Fehler behandeln, ohne dass das gesamte Skript abbricht.

```
try {
    # Code, der einen Fehler verursachen könnte
    Get-Content "nicht_existierende_datei.txt"
} catch {
    # Fehlerbehandlung
    Write-Host "Ein Fehler ist aufgetreten: $_"
}
```

Fortgeschrittenes Skripting

Fehlerbehandlung und Debugging

2. $Error-Variable: PowerShell speichert alle Fehler in der `$Error`-Variable. Du kannst diese Variable verwenden, um Informationen über die letzten Fehler abzurufen.

3. Set-StrictMode: Mit diesem Befehl kannst du strikte Regeln für dein Skript festlegen, was dir hilft, potenzielle Fehler frühzeitig zu erkennen.

Fortgeschrittenes Skripting

Fehlerbehandlung und Debugging

Debugging ist der Prozess, bei dem du versuchst, die Ursache eines Fehlers zu finden. In PowerShell gibt es einige nützliche Werkzeuge:

1. Set-PSBreakpoint: Damit kannst du Haltepunkte in deinem Skript setzen, an denen die Ausführung stoppt, sodass du den aktuellen Zustand der Variablen und den Programmfluss überprüfen kannst.

2. Write-Debug: Mit diesem Befehl kannst du Debugging-Nachrichten in deinem Skript ausgeben, die dir helfen, den Fluss und die Werte während der Ausführung zu verfolgen.

3. Verbose-Parameter: Viele Cmdlets unterstützen den `-Verbose`-Parameter, der zusätzliche Informationen über die Ausführung bereitstellt.

Fortgeschrittenes Skripting

Fehlerbehandlung und Debugging

Zusammengefasst:

Fehlerbehandlung helfen, mit Problemen umzugehen, während Debugging hilft, die Ursachen dieser Probleme zu finden. Beide Techniken sind entscheidend, um effektive und zuverlässige PowerShell-Skripte zu schreiben.

Fortgeschrittenes Skripting

Arbeiten mit Skript-Parameter und Argumenten

In PowerShell kannst du Skripte erstellen, die Parameter und Argumente verwenden, um die Funktionalität zu erweitern und die Wiederverwendbarkeit zu erhöhen. Lass uns das Schritt für Schritt durchgehen.

Was sind Parameter und Argumente?

- **Parameter:** Das sind die Variablen, die du in deinem Skript definierst. Sie dienen dazu, Eingabewerte zu akzeptieren, die das Skript benötigt, um seine Aufgabe auszuführen.

- **Argumente:** Das sind die Werte, die du beim Aufruf des Skripts übergibst. Diese Werte werden den Parametern zugewiesen.

Fortgeschrittenes Skripting

Arbeiten mit Skript-Parameter und Argumenten

Ein einfaches Beispiel

Stell dir vor, du möchtest ein Skript erstellen, das einen Namen entgegennimmt und eine Begrüßung ausgibt. Hier ist, wie du das machen könntest:

1. Skript erstellen: Erstelle eine Datei mit der Endung `.ps1`, z.B. `Begruessung.ps1`.

Fortgeschrittenes Skripting

Arbeiten mit Skript-Parameter und Argumenten

2. Parameter definieren: In deinem Skript definierst du einen Parameter für den Namen.

```
param (
    [string]$Name
)

Write-Output "Hallo, $Name! Willkommen!"
```

.

3. Skript ausführen: Wenn du das Skript ausführen möchtest, gibst du den Namen als Argument an:

```
.\Begruessung.ps1 -Name "Max"
```

Das Skript gibt dann aus:

```
Hallo, Max! Willkommen!
```

Fortgeschrittenes Skripting

Arbeiten mit Skript-Parameter und Argumenten

Vorteile der Verwendung von Parametern

- Flexibilität: Du kannst das Skript mit verschiedenen Werten aufrufen, ohne den Code ändern zu müssen.
- Lesbarkeit: Es ist klar, welche Eingaben das Skript erwartet.
- Wiederverwendbarkeit: Du kannst das Skript in verschiedenen Szenarien verwenden, indem du einfach andere Argumente übergibst.

Fazit:

Das Arbeiten mit Skript-Parametern und Argumenten in PowerShell ist eine großartige Möglichkeit, deine Skripte dynamischer und benutzerfreundlicher zu gestalten. Es ermöglicht dir, Skripte zu schreiben, die vielseitig einsetzbar sind und leicht angepasst werden können. Wenn du weitere Fragen hast oder mehr Beispiele benötigst, lass es mich wissen!

02 Automatisierung und Scheduling

Automatisierung und Scheduling

Themen

1. Verwendung von Task Scheduler
2. Erstellen von geplanten Aufgaben mit PowerShell
3. Automatisierung von wiederkehrenden Aufgaben

Automatisierung und Scheduling

Verwendung von Task Scheduler

Der Task Scheduler in PowerShell ist ein nützliches Tool, mit dem du automatisierte Aufgaben auf deinem Computer planen und verwalten kannst. Hier ist eine einfache Erklärung, wie du ihn verwenden kannst:

1. **Was ist der Task Scheduler?**

Der Task Scheduler ist ein Programm, das es dir ermöglicht, bestimmte Aufgaben zu bestimmten Zeiten oder bei bestimmten Ereignissen automatisch auszuführen. Zum Beispiel kannst du ein Skript planen, das jeden Tag um 8 Uhr morgens ausgeführt wird.

Automatisierung und Scheduling

Verwendung von Task Scheduler

2. **Wie kannst du ihn in PowerShell nutzen?**

In PowerShell kannst du den Task Scheduler über verschiedene Cmdlets (Befehle) ansprechen. Einige der häufigsten Cmdlets sind:

- `New-ScheduledTask`: Damit kannst du eine neue geplante Aufgabe erstellen.
- `Register-ScheduledTask`: Damit registrierst du die geplante Aufgabe im Task Scheduler.
- `Get-ScheduledTask`: Damit kannst du eine Liste der geplanten Aufgaben abrufen.
- `Unregister-ScheduledTask`: Damit kannst du eine geplante Aufgabe löschen.

Automatisierung und Scheduling

Verwendung von Task Scheduler

2. **Wie kannst du ihn in PowerShell nutzen?**

3. Ein einfaches Beispiel:
Angenommen, du möchtest ein Skript namens
`MeinSkript.ps1`
jeden Montag um 10 Uhr ausführen. Du würdest die folgenden Schritte in PowerShell ausführen:

```
$action = New-ScheduledTaskAction -Execute "PowerShell.exe" -Argument"-File C:\Pfad\Zu\MeinSkript.ps1"
$trigger = New-ScheduledTaskTrigger -Weekly -At "10:00AM" -DaysOfWeek Monday
Register-ScheduledTask -Action $action -Trigger $trigger -TaskName "MeinWöchentlicherTask"
```

4. **Zusammenfassung:**

Der Task Scheduler in PowerShell ist ein praktisches Werkzeug, um Aufgaben zu automatisieren. Mit ein paar einfachen Befehlen kannst du Aufgaben erstellen, die zu bestimmten Zeiten oder unter bestimmten Bedingungen ausgeführt werden.

Automatisierung und Scheduling

Erstellen von geplanten Aufgaben mit PowerShell

Das Erstellen von geplanten Aufgaben mit PowerShell ist eine praktische Möglichkeit, um bestimmte Skripte oder Programme automatisch zu bestimmten Zeiten oder bei bestimmten Ereignissen auszuführen.

Hier ist eine einfache Schritt-für-Schritt-Anleitung:

1. **Öffne PowerShell:**
Du kannst dies tun, indem du im Startmenü nach "PowerShell" suchst und es als Administrator ausführst.

2. **Verwende das Cmdlet**

`New-ScheduledTask`

: Mit diesem Cmdlet kannst du eine neue geplante Aufgabe erstellen.

Automatisierung und Scheduling

Erstellen von geplanten Aufgaben mit PowerShell

Hier ist ein einfaches Beispiel:

```
$action = New-ScheduledTaskAction -Execute "notepad.exe"
$trigger = New-ScheduledTaskTrigger -At 9am -Daily
$principal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType ServiceAccount
Register-ScheduledTask -Action $action -Trigger $trigger -Principal $principal -TaskName "ÖffneNotepad"
```

In diesem Beispiel:

- `New-ScheduledTaskAction` definiert, was die Aufgabe tun soll (in diesem Fall Notepad öffnen).
- `New-ScheduledTaskTrigger` legt fest, wann die Aufgabe ausgeführt wird (hier täglich um 9 Uhr).
- `New-ScheduledTaskPrincipal` gibt an, unter welchem Benutzerkonto die Aufgabe ausgeführt wird (hier als SYSTEM).
- `Register-ScheduledTask` registriert die Aufgabe mit einem Namen.

Automatisierung und Scheduling

Erstellen von geplanten Aufgaben mit PowerShell

3. **Überprüfe die geplante Aufgabe:**

Du kannst die geplanten Aufgaben in der Aufgabenplanung von Windows überprüfen, um sicherzustellen, dass alles korrekt eingerichtet ist.

Das war's! Mit diesen einfachen Schritten kannst du geplante Aufgaben mit PowerShell erstellen.

Automatisierung und Scheduling

Automatisierung von wiederkehrenden Aufgaben

Automatisierung von wiederkehrenden Aufgaben in PowerShell bedeutet, dass du bestimmte Aufgaben, die du regelmäßig durchführen musst, automatisieren kannst, sodass du sie nicht manuell ausführen musst.

Stell dir vor, du musst jeden Tag eine Datei kopieren, einen Bericht erstellen oder Systemupdates durchführen. Anstatt das alles von Hand zu machen, kannst du ein Skript in PowerShell schreiben, das diese Aufgaben für dich erledigt.

Automatisierung und Scheduling

Automatisierung von wiederkehrenden Aufgaben

Hier sind die grundlegenden Schritte, um das zu tun:

1. **Skript erstellen:** Du schreibst ein PowerShell-Skript, das die gewünschten Befehle enthält. Zum Beispiel könnte das Skript so aussehen, dass es eine Datei kopiert oder einen Dienst neu startet.

2. **Planen:** Mit dem Windows Task Scheduler kannst du festlegen, wann und wie oft dein Skript ausgeführt werden soll. Du kannst es täglich, wöchentlich oder zu einem bestimmten Zeitpunkt planen.

3. **Ausführen:** Der Task Scheduler führt dein Skript automatisch aus, ohne dass du etwas tun musst. So sparst du Zeit und stellst sicher, dass die Aufgaben regelmäßig erledigt werden.

Das ist eine einfache und effektive Möglichkeit, um Routineaufgaben zu automatisieren und deinen Arbeitsalltag zu erleichtern!

03 Verwaltung von Systemen und Netzwerken

Verwaltung von Systemen und Netzwerken

Inhalt

1. Remote Management mit PowerShell Remoting
2. Arbeiten mit WMI und CIM
3. Netzwerkadministration und -überwachung

Verwaltung von Systemen und Netzwerken

Remote Management mit PowerShell Remoting

Remote Management mit PowerShell Remoting ist eine Möglichkeit, Computer über das Netzwerk zu steuern und zu verwalten, ohne direkt vor dem Computer zu sitzen. Stell dir vor, du hast mehrere Computer in einem Büro, und du möchtest auf jedem von ihnen etwas ändern oder überprüfen. Anstatt zu jedem Computer zu gehen, kannst du das ganz bequem von deinem eigenen Computer aus machen.

Verwaltung von Systemen und Netzwerken

Remote Management mit PowerShell Remoting

Schritt 1: **Voraussetzungen prüfen**

Bevor du mit PowerShell Remoting arbeiten kannst, stelle sicher, dass du die folgenden

Voraussetzungen erfüllst:

- Du benötigst Windows 10 oder Windows Server 2012 und höher.
- Du solltest Administratorrechte auf dem Computer haben, den du verwalten möchtest.

Verwaltung von Systemen und Netzwerken

Remote Management mit PowerShell Remoting

Schritt 2: **PowerShell Remoting aktivieren**
Um PowerShell Remoting zu aktivieren, öffne PowerShell mit Administratorrechten und führe den folgenden Befehl aus:

```
Enable-PSRemoting -Force
```

Dieser Befehl konfiguriert die notwendigen Einstellungen und öffnet die erforderlichen Firewall-Ports.

Schritt 3: **Überprüfen der Remoting-Einstellungen**
Um sicherzustellen, dass PowerShell Remoting korrekt aktiviert wurde, kannst du den folgenden Befehl verwenden:

```
Get-PSRemotingConfiguration
```

Dieser Befehl zeigt dir die aktuellen Remoting-Einstellungen an.

Verwaltung von Systemen und Netzwerken

Remote Management mit PowerShell Remoting

Schritt 4: **Remote-Verbindung herstellen**
Um eine Verbindung zu einem anderen Computer herzustellen, verwende den

`Enter-PSSession`

Befehl.

Ersetze

`ComputerName`

durch den Namen oder die IP-Adresse des Zielcomputers:

```
Enter-PSSession -ComputerName ComputerName -Credential (Get-Credential)
```

Ein Fenster wird geöffnet, in dem du die Anmeldeinformationen für den Zielcomputer eingeben kannst.

Verwaltung von Systemen und Netzwerken

Remote Management mit PowerShell Remoting

Schritt 7: **Remote-Befehle ausführen (optional)**
Wenn du Befehle auf einem Remote-Computer ausführen möchtest, ohne eine interaktive Sitzung zu starten, kannst du den `Invoke-Command` Befehl verwenden:

```
Invoke-Command -ComputerName ComputerName -ScriptBlock { Get-Process }
```

Dieser Befehl führt den angegebenen Skriptblock auf dem Remote-Computer aus und gibt die Ergebnisse zurück.

Das war eine einfache Schritt-für-Schritt-Anleitung, um PowerShell Remoting einzurichten und zu nutzen. Mit diesen Grundlagen kannst du effizient Remote Management durchführen.

Verwaltung von Systemen und Netzwerken

Arbeiten mit WMI und CIM

WMI (Windows Management Instrumentation) und CIM (Common Information Model) sind beides Technologien, die es ermöglichen, Informationen über die Hardware und Software eines Computers zu sammeln und zu verwalten. Beide können in PowerShell genutzt werden, um verschiedene Aufgaben auf einem Windows-System zu erledigen, wie zum Beispiel das Abrufen von Systeminformationen, das Verwalten von Prozessen oder das Konfigurieren von Hardwarekomponenten.

Verwaltung von Systemen und Netzwerken

Arbeiten mit WMI und CIM

Was ist WMI?

WMI ist eine Sammlung von Systemdaten und -funktionen, die es ermöglichen, Informationen über Windows-Computer zu erhalten, z. B. über das Betriebssystem, die Hardware, Netzwerkeinstellungen oder installierte Software. Mit PowerShell kannst du WMI verwenden, um diese Informationen zu lesen oder zu ändern.

Was ist CIM?

CIM ist ein standardisiertes Modell für das Abrufen von Systeminformationen. CIM ist ähnlich wie WMI, aber moderner und auch plattformübergreifend (d.h., es kann nicht nur auf Windows, sondern auch auf anderen Systemen verwendet werden). CIM wird oft bevorzugt, weil es flexibler und effizienter ist.

Verwaltung von Systemen und Netzwerken

Arbeiten mit WMI und CIM

Arbeiten mit PowerShell

In PowerShell kannst du sowohl mit WMI als auch mit CIM arbeiten. Die wichtigsten Cmdlets (Befehle), die dafür verwendet werden, sind:

1. WMI in PowerShell

Get-WmiObject: Wird verwendet, um Informationen aus WMI abzurufen. Beispiel:

```
Get-WmiObject -Class Win32_OperatingSystem
```

Dieser Befehl zeigt dir Informationen zum Betriebssystem.

Verwaltung von Systemen und Netzwerken

Arbeiten mit WMI und CIM

2. CIM in PowerShell

Get-CimInstance: Ist der CIM-Äquivalent von „Get-WmiObject“ und wird für dieselben Aufgaben verwendet, aber auf einer moderneren Technologie basierend. Beispiel:

```
Get-CimInstance -ClassName Win32_OperatingSystem
```

Auch dieser Befehl zeigt dir Informationen zum Betriebssystem, aber auf eine Weise, die mit anderen Plattformen kompatibel ist.

Unterschied zwischen WMI und CIM

- **WMI**: Älter, speziell für Windows, aber auch leistungsfähig.
- **CIM**: Neuer, plattformübergreifend, effizienter, und wird in Zukunft häufiger verwendet werden.

Verwaltung von Systemen und Netzwerken

Arbeiten mit WMI und CIM

Praktisches Beispiel:

1. **WMI**: Informationen über Prozessor abrufen:

```
Get-WmiObject -Class Win32_Processor
```

1. **CIM**: Dasselbe mit CIM:

```
Get-CimInstance -ClassName Win32_Processor
```

Beide Befehle geben dir Informationen zum Prozessor, aber „Get-CimInstance“ arbeitet effizienter und ist die bevorzugte Wahl.

Beide Technologien dienen dazu, Systeminformationen zu sammeln, aber CIM ist die modernere, zukunftssichere Variante, während WMI in älteren Windows-Versionen immer noch weit verbreitet ist.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

Netzwerkadministration und -überwachung mit PowerShell bedeutet, dass du mit PowerShell-Commands (Cmdlets) dein Netzwerk überwachen, konfigurieren und Probleme diagnostizieren kannst. PowerShell bietet viele Tools, die dir helfen, Netzwerkgeräte wie Router, Switches und Computer zu verwalten und zu überwachen, ohne dafür extra Software installieren zu müssen.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

1. Netzwerkverbindungen prüfen

Mit PowerShell kannst du schnell prüfen, ob dein Computer mit dem Netzwerk verbunden ist.

Beispiel: **Testen, ob du eine Website erreichen kannst:**

```
Test-Connection www.google.com
```

Dieser Befehl funktioniert ähnlich wie der klassische „Ping“-Befehl und zeigt dir, ob du eine Verbindung zu einer Website oder einem anderen Gerät im Netzwerk herstellen kannst.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

2. IP-Adresse herausfinden

Manchmal musst du die IP-Adresse deines Computers oder eines anderen Geräts im Netzwerk herausfinden.

Beispiel: **IP-Adresse des Computers anzeigen:**

```
Get-NetIPAddress
```

Dieser Befehl zeigt dir alle IP-Adressen an, die deinem Computer zugewiesen sind (zum Beispiel IPv4 und IPv6).

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

3. Netzwerkkarten verwalten

Du kannst die Netzwerkkarten (Adapter) deines Computers anzeigen oder konfigurieren.

Beispiel: **Alle Netzwerkadapter anzeigen:**

```
Get-NetAdapter
```

Dieser Befehl gibt dir eine Liste aller Netzwerkadapter zurück, die auf deinem Computer installiert sind, z.B. WLAN oder Ethernet.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

4. DNS-Server anzeigen und konfigurieren

Du kannst auch die DNS-Server anzeigen, die dein Computer verwendet, oder sie ändern.

Beispiel: **DNS-Server anzeigen:**

```
Get-DnsClientServerAddress
```

Dieser Befehl zeigt dir die DNS-Server an, die dein Computer aktuell verwendet.

Beispiel: **DNS-Server ändern:**

```
Set-DnsClientServerAddress -InterfaceIndex 12 -ServerAddresses
("8.8.8.8", "8.8.4.4")
```

Hiermit änderst du den DNS-Server auf die Google DNS-Server.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

5. Netzwerkfreigaben verwalten

Mit PowerShell kannst du auch Netzwerkfreigaben (wie geteilte Ordner) erstellen oder verwalten.

Beispiel: Alle freigegebenen Ordner anzeigen:

```
Get-SmbShare
```

Dieser Befehl zeigt dir alle freigegebenen Ordner auf deinem Computer.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

6. Netzwerkverkehr überwachen

Mit PowerShell kannst du den Netzwerkverkehr überwachen und Probleme diagnostizieren.

Beispiel: Netzwerkstatistiken anzeigen:

```
Get-NetStat
```

Dieser Befehl zeigt dir die aktuellen Netzwerkverbindungen und -statistiken auf deinem Computer an, z. B. offene Verbindungen und die zugehörigen Ports.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

7. Netzwerkprobleme diagnostizieren

PowerShell kann dir auch helfen, Netzwerkprobleme zu finden und zu beheben.

Beispiel: Netzwerkkonfiguration zurücksetzen:

```
Reset-NetAdapterAdvancedProperty -Name "Ethernet" -DisplayName "Speed & Duplex"
```

Mit diesem Befehl kannst du eine Netzwerkeinstellung auf dem Adapter zurücksetzen.

Verwaltung von Systemen und Netzwerken

Netzwerkadministration und -überwachung

Zusammengefasst:

PowerShell bietet viele nützliche Befehle, mit denen du dein Netzwerk verwalten und überwachen kannst, ohne auf grafische Benutzeroberflächen angewiesen zu sein. Du kannst:

- Verbindungen testen
- IP-Adressen und DNS-Server anzeigen
- Netzwerkkarten verwalten
- Netzwerkfreigaben anzeigen
- Den Netzwerkverkehr überwachen und Probleme diagnostizieren.

Es ist ein mächtiges Tool, besonders für Administratoren, die viele Computer oder Netzwerkeinstellungen automatisiert verwalten müssen.

04
Sicherheit und Best Practices

Verwaltung von Systemen und Netzwerken

Themen

1. Verwendung von Code-Signing
2. Best Practices für die Skripterstellung

Verwaltung von Systemen und Netzwerken

Verwendung von Code-Signing

Code-Signing in PowerShell bedeutet, deine Skripte mit einer digitalen Unterschrift zu versehen. Diese Unterschrift bestätigt, dass das Skript von einer vertrauenswürdigen Quelle kommt und seit der Signierung nicht verändert wurde. Code-Signing hilft also dabei, sicherzustellen, dass niemand das Skript verändert hat, um es z.B. einen Exploit auszuführen, und dass du als Ersteller des Skripts erkannt wirst.

Verwaltung von Systemen und Netzwerken

Verwendung von Code-Signing

Warum ist Code-Signing wichtig?

1. **Vertrauen schaffen**: Wenn ein Skript signiert ist, können andere Benutzer oder Systeme sicherstellen, dass es von einer vertrauenswürdigen Quelle stammt und nicht von jemandem verändert wurde, und somit keinen schadhaften Code enthalten.
2. **Sicherheit erhöhen**: Viele Systeme, einschließlich PowerShell, erlauben das Ausführen von Skripten nur, wenn sie signiert sind (z. B. bei der Ausführungsrichtlinie „AllSigned“ oder „RemoteSigned“). Dadurch werden unsichere oder unbekannte Skripte blockiert.
3. **Verhindert das Ausführen von schadhafter Software**: Ein signiertes Skript kann nicht einfach von einem Dritten geändert werden. Falls das Skript nach der Signierung verändert wird, wird die digitale Unterschrift ungültig und das Skript wird nicht ausgeführt.

Verwaltung von Systemen und Netzwerken

Verwendung von Code-Signing

Wie funktioniert Code-Signing?

Um Code-Signing in PowerShell zu verwenden, musst du dein Skript mit einem **Zertifikat** signieren. Ein Zertifikat ist eine Art digitale „ID“, die dir gehört. Es bestätigt deine Identität und wird von einer vertrauenswürdigen Stelle ausgestellt (z. B. einer Zertifizierungsstelle).

Schritte, um ein PowerShell-Skript zu signieren:

- **Erstelle ein Zertifikat**: Du brauchst ein Zertifikat, um deine Skripte zu signieren. Du kannst ein selbstsigniertes Zertifikat für Tests erstellen oder ein Zertifikat von einer Zertifizierungsstelle erwerben. Um ein selbstsigniertes Zertifikat zu erstellen, kannst du in PowerShell folgenden Befehl verwenden:

```
New-SelfSignedCertificate -Type CodeSigning -Subject "CN=MeinName"
-KeyExportPolicy Exportable -KeyLength 2048 -KeyAlgorithm RSA -
CertStoreLocation "Cert:\CurrentUser\My"
```

Verwaltung von Systemen und Netzwerken

Verwendung von Code-Signing

- **Finde das Zertifikat**: Nachdem du das Zertifikat erstellt hast, musst du es in deinem Zertifikatspeicher finden. Du kannst es mit folgendem Befehl anzeigen:

  ```
  Get-ChildItem -Path Cert:\CurrentUser\My
  ```

- **Signiere das Skript**: Du kannst das Skript mit dem Zertifikat signieren. Hierzu verwendest du das Cmdlet `Set-AuthenticodeSignature`. Zum Beispiel:

  ```
  Set-AuthenticodeSignature -FilePath "C:\Pfad\zu\deinem\Skript.ps1"
  -Certificate "MeinZertifikat"
  ```

  Dieser Befehl signiert das Skript `Skript.ps1` mit deinem Zertifikat.

Verwaltung von Systemen und Netzwerken

Verwendung von Code-Signing

- **Überprüfe die Signatur**: Du kannst prüfen, ob ein Skript korrekt signiert wurde, indem du den Befehl `Get-AuthenticodeSignature` verwendest:

  ```
  Get-AuthenticodeSignature "C:\Pfad\zu\deinem\Skript.ps1"
  ```

Wie funktioniert die Ausführung signierter Skripte?

Wenn du in PowerShell eine **strenge Ausführungsrichtlinie** wie „**AllSigned**“ verwendest, werden nur signierte Skripte ausgeführt. Wenn du versuchst, ein unsigniertes Skript auszuführen, wird eine Fehlermeldung angezeigt.

Zum Beispiel, wenn du versuchst, ein unsigniertes Skript auszuführen, und die Richtlinie „AllSigned“ festgelegt ist, bekommst du eine Warnung oder eine Fehlermeldung, dass das Skript nicht ausgeführt werden kann.

Verwaltung von Systemen und Netzwerken

Verwendung von Code-Signing

Vorteile von Code-Signing:

- **Vertrauen**: Andere können sicher sein, dass dein Skript aus einer vertrauenswürdigen Quelle stammt.
- **Sicherheit**: Du schützt dein Skript davor, dass es nach der Signierung verändert wird, da jede Veränderung die Signatur ungültig macht.
- **Zertifikate**: Mit einem gültigen Zertifikat kannst du sicherstellen, dass dein Skript auch in Umgebungen mit strengen Sicherheitsrichtlinien ausgeführt werden kann.

Zusammenfassung:

Code-Signing bedeutet, dass du ein PowerShell-Skript mit einer digitalen Unterschrift versiehst. Diese Signatur stellt sicher, dass das Skript von einer vertrauenswürdigen Quelle stammt und nicht verändert wurde. Du benötigst dazu ein Zertifikat, das du entweder selbst erstellen oder von einer Zertifizierungsstelle erhalten kannst. Code-Signing erhöht die Sicherheit und sorgt dafür, dass nur vertrauenswürdige Skripte auf deinem System ausgeführt werden.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

Wenn du PowerShell-Skripte erstellst, gibt es einige Best Practices, die dir helfen, sauberen, sicheren und wartbaren Code zu schreiben. Diese Best Practices sorgen nicht nur dafür, dass dein Skript zuverlässig funktioniert, sondern auch, dass es leichter zu verstehen und zu pflegen ist, besonders wenn du es später wieder verwenden oder mit anderen teilen möchtest.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

1. Verwende aussagekräftige Namen für Variablen und Funktionen

Gute Namen machen deinen Code verständlicher. Wenn jemand (oder auch du selbst) später das Skript liest, sollte er sofort wissen, was eine Variable oder Funktion macht.

- **Schlechte Variable**: `$x`, `$data`
- **Bessere Variable**: `$userList`, `$serverStatus`

Vermeide kryptische Namen und versuche, Namen zu wählen, die klar beschreiben, was sie repräsentieren.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

2. Kommentiere deinen Code

Kommentare sind Erklärungen, die du in deinem Code hinterlässt, um zu beschreiben, was der Code tut. Auch wenn du denkst, dass du deinen Code verstehst, können Kommentare hilfreich sein, wenn du nach Wochen oder Monaten wieder darauf zugreifst.

- **Beispiel**:

```
# Diese Funktion prüft, ob der Benutzer bereits existiert
function Check-UserExists {
        # Code hier
}
```

Verwende Kommentare, um schwierige oder komplexe Teile des Codes zu erklären.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

3. Vermeide magische Zahlen und Strings

„Magische Zahlen“ oder „magische Strings“ sind Werte, die im Code vorkommen, aber nicht erklärt werden, warum sie verwendet werden. Statt direkt Zahlen oder Strings zu verwenden, solltest du sie als Konstanten oder Variablen definieren.

- **Beispiel**:

```
# Statt 100 direkt zu verwenden:

$timeoutValue = 100  # Timeout in Sekunden
```

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

4. Fehlerbehandlung einbauen

PowerShell-Skripte können auf Fehler stoßen, z. B. wenn eine Datei nicht gefunden wird oder der Netzwerkzugriff fehlschlägt. Es ist wichtig, diese Fehler abzufangen und sinnvoll zu behandeln.

- **Beispiel für Fehlerbehandlung**:

```
try {
    # Versuch, eine Datei zu öffnen
    $fileContent = Get-Content "C:\beispiel.txt"
}
catch {
    # Fehlerbehandlung
    Write-Host "Fehler: Die Datei konnte nicht geöffnet werden."
}
```

Durch Fehlerbehandlung kannst du verhindern, dass das Skript einfach abstürzt, und kannst dem Benutzer eine klare Fehlermeldung anzeigen.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

5. Verwende Funktionen und Module

Anstatt immer den gleichen Code zu wiederholen, solltest du häufig genutzte Logik in Funktionen oder Module auslagern. Das macht deinen Code modularer und wiederverwendbar.

- **Beispiel**:

```
# Funktion zur Prüfung, ob eine Datei existiert
function Test-FileExists {
    param([string]$filePath)
    if (Test-Path $filePath) {
        return $true
    }
    else {
        return $false
    }
}
```

Funktionen und Module fördern die Wiederverwendbarkeit und helfen, den Code übersichtlicher zu halten.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

6. Nutze `-WhatIf` und `-Confirm` für sicherheitskritische Aktionen

Wenn dein Skript Änderungen an Systemen vornimmt (z. B. das Löschen von Dateien oder das Stoppen von Diensten), ist es wichtig, sicherzustellen, dass der Benutzer der Änderung zustimmt, bevor sie tatsächlich ausgeführt wird.

- **Beispiel für -Whatlf**:

```
Remove-Item "C:\WichtigeDatei.txt" -WhatIf
```

Der `-WhatIf`-Schalter zeigt dir nur an, was passieren würde, ohne die Aktion auszuführen.

- **Beispiel für -Confirm**:

```
Remove-Service "BeispielDienst" -Confirm
```

Der `-Confirm`-Schalter fordert den Benutzer zu einer Bestätigung auf, bevor die Aktion durchgeführt wird.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

7. Verwende Pipelining und Cmdlets

PowerShell ist besonders stark, wenn du das Pipelining verwendest. Dadurch kannst du das Ergebnis eines Cmdlets direkt an ein anderes Cmdlet weitergeben, ohne Zwischenspeicher zu verwenden.

- **Beispiel**:

```
Get-Process | Where-Object { $_.CPU -gt 100 }
```

Hier filterst du alle Prozesse, deren CPU-Nutzung über 100 liegt, direkt durch das Pipeline-System. Dies macht den Code eleganter und effizienter.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

8. Setze eine geeignete Ausführungsrichtlinie

Stelle sicher, dass deine Ausführungsrichtlinie (Execution Policy) korrekt eingestellt ist. Wenn du Skripte nur aus sicheren Quellen ausführen möchtest, solltest du „RemoteSigned“ oder „AllSigned“ verwenden. Dadurch wird verhindert, dass unsichere Skripte ausgeführt werden.

- **Beispiel**:

```
Set-ExecutionPolicy RemoteSigned
```

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

9. Dokumentiere dein Skript

Erstelle am Anfang deines Skripts eine kurze Beschreibung, was das Skript tut und wie es verwendet wird. Wenn andere Personen dein Skript verwenden sollen, ist eine Dokumentation sehr hilfreich.

- **Beispiel**:

```
# Skript: Prüft den Status von Diensten auf einem Remote-Server

# Verwendung: .\Check-ServiceStatus.ps1 -ComputerName "Server01"
```

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

10. Testen und Debuggen

Bevor du dein Skript in einer Produktionsumgebung einsetzt, solltest du es gründlich testen. Verwende dafür PowerShell’s Debugging-Tools, um sicherzustellen, dass alles funktioniert.

- **Beispiel**:

```
Set-PSDebug -Trace 1
```

Dadurch kannst du die Ausführung deines Skripts Schritt für Schritt nachverfolgen und Fehler leichter finden.

Verwaltung von Systemen und Netzwerken

Best Practices für die Skripterstellung

Zusammenfassung:

Die wichtigsten **Best Practices** für PowerShell-Skripte sind:

- **Aussagekräftige Namen** für Variablen und Funktionen.
- **Kommentare**, um den Code verständlicher zu machen.
- **Vermeidung von magischen Zahlen** und Strings, durch deren Definition als Variablen.
- **Fehlerbehandlung** einbauen, um Abstürze zu vermeiden.
- **Verwendung von Funktionen und Modulen**, um Wiederverwendbarkeit und Übersichtlichkeit zu erhöhen.
- **Sicherheitsmaßnahmen** wie `-WhatIf` und `-Confirm`.
- **Pipelining** nutzen, um den Code effizienter zu gestalten.
- **Testen und Debuggen** vor der Verwendung in Produktionsumgebungen.

Indem du diese Best Practices beachtest, schreibst du nicht nur funktionale, sondern auch wartbare und sichere PowerShell-Skripte.

DANKE!

Gibt es noch Fragen?

CloudCommand