Skrypty PowerShell

Struktura skryptu, audyty Active Directory, zadania, usługi i poczta.

- Skrypty PowerShell: struktura i dobre praktyki
- Zadania zaplanowane, usługi i procesy
- Wysyłanie poczty ze skryptu

Skrypty PowerShell: struktura i dobre praktyki

Poziom: średniozaawansowany **Dotyczy: Windows PowerShell 5.1, PowerShell 7+**

Czas: ~35 min

Od pliku `.ps1` z trzema poleceniami do skryptu, który bezpiecznie uruchomi ktoś inny, o trzeciej w nocy, bez nadzoru — parametry, tryb ścisły, obsługa błędów i szkielet gotowy do skopiowania.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
TCP	Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

Fiszka

TYLKO ODCZYT

Zdefiniowanie parametrów skryptu: param()

```
param(
    [string]$NazwaSerwera,
    [int]$NumerPortu = 443
)
```

> **Pojęcie: param().** Blok na samym początku pliku `.ps1` (przed jakąkolwiek inną instrukcją wykonywalną, komentarze dozwolone) deklarujący, jakie argumenty skrypt przyjmuje z linii wywołania — `.\skrypt.ps1 -NazwaSerwera wsus01 -NumerPortu 8443`. Bez `param()` skrypt wciąż może odwoływać się do argumentów przez `$args`, ale traci nazwane parametry, walidację typu i podpowiadanie składni w edytorze.

🔒 TYLKO ODCZYT

Wymuszenie podania parametru: [Parameter(Mandatory)]

```
param(
    [Parameter(Mandatory)]
    [string]$NazwaSerwera
)
```

Skrypt uruchomiony bez `-NazwaSerwera` nie wystartuje z pustą albo domyślną wartością — PowerShell sam zapyta interaktywnie o brakujący parametr, a w sesji bez możliwości interakcji (na przykład w zadaniu zaplanowanym) zwróci błąd zamiast ciągnąć dalej z niekompletnymi danymi.

Przykład:

```
cmdlet skrypt.ps1 at command pipeline position 1
Supply values for the following parameters:
NazwaSerwera:
```

🔒 TYLKO ODCZYT

Ograniczenie parametru do zamkniętej listy wartości: ValidateSet

```
param(
    [ValidateSet("Restart", "Stop", "Start")]
    [string]$Akcja = "Start"
)
```

Skrypt odrzuca dowolną wartość spoza podanej listy jeszcze przed wykonaniem pierwszej linii kodu — literówka w nazwie akcji (`"Restrat"`) kończy się jasnym komunikatem błędu walidacji, zamiast cichym przejściem do gałęzi `else`, której nikt nie napisał na taki przypadek.

Przykład:

```
Cannot validate argument on parameter 'Akcja'. The argument "Restrat" does not belong to
the set
"Restart,Stop,Start" specified by the ValidateSet attribute.
```

TYLKO ODCZYT

Walidacja zakresu wartości liczbowej: ValidateRange

```
param(
    [ValidateRange(1, 65535)]
    [int]$NumerPortu
)
```

Odrzucamy wartości spoza sensownego zakresu numeru portu TCP/UDP na etapie parsowania parametrów, zamiast dowiadywać się o błędzie dopiero przy próbie faktycznego połączenia się na port `0` albo `99999`.

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

TYLKO ODCZYT

Walidacja przez własny blok skryptu: ValidateScript

```
param(
    [ValidateScript({ Test-Path $_ })]
    [string]$SciezkaDoPliku
)
```

Sprawdzamy dowolny, własny warunek na wartości parametru — tutaj: czy podana ścieżka w ogóle istnieje na dysku, zanim skrypt spróbuje z niej cokolwiek odczytać.

TYLKO ODCZYT

Deklaracja typu parametru wymuszająca konwersję

```
param(
    [datetime]$DataOd
)
```

Wpisanie `-DataOd "2026-01-15"` na wejściu skryptu automatycznie konwertuje tekst na obiekt `DateTime` — dalszy kod operuje na prawdziwej dacie z dostępnymi metodami i porównaniami, nie na tekście wymagającym ręcznego parsowania.

🔍 TYLKO ODCZYT

Przekazanie zestawu parametrów przez tablicę haszującą: splatting

```
$parametry = @{
    ComputerName = "NazwaKomputera"
    Credential   = $poswiadczenia
    ScriptBlock  = { Get-Service }
}
Invoke-Command @parametry
```

> **Pojęcie: splatting.** Zamiast wypisywać każdy parametr wywołania osobno w jednej długiej linii, składamy je w tablicę haszującą i przekazujemy całość znakiem `@` zamiast `$` przed nazwą zmiennej — czytelniejsze przy poleceniach z wieloma parametrami i wygodne, gdy część parametrów jest warunkowa (dopisywana do tablicy tylko czasem, przed wywołaniem).

⚠️ ZMIENIA SYSTEM

Włączenie trybu ścisłego: Set-StrictMode

```
Set-StrictMode -Version Latest
```

> **Pojęcie: tryb ścisły w PowerShellu.** Odpowiednik `set -euo pipefail` z basha, opisanego na stronie Tryb ścisły: set -euo pipefail — z tą różnicą, że `Set-StrictMode` skupia się na innym zestawie pułapek niż bashowy `set -u`. Włączony na `-Version Latest`, wymusza błąd przy odwołaniu do nieistniejącej właściwości obiektu, przy użyciu niezainicjalizowanej zmiennej i przy niepoprawnym wywołaniu metody — bez niego PowerShell po cichu zwraca `$null` w wielu z tych sytuacji, co maskuje literówkę w nazwie właściwości równie skutecznie jak brak `set -u` maskuje literówkę w nazwie zmiennej basha.

Przykład błędu wyłapanego przez tryb ścisły:

```
InvalidOperation: The property 'StatuS' cannot be found on this object.
```

🔍 TYLKO ODCZYT

Ustawienie zachowania przy błędach niekrytycznych: $ErrorActionPreference

```
$ErrorActionPreference = "Stop"
```

> **Pojęcie: błąd przerywający i nieprzerywający.** PowerShell rozróżnia dwa rodzaje błędów — przerywające (`terminating`), które natychmiast zatrzymują bieżący blok kodu, i nieprzerywające (`non-terminating`), które domyślnie tylko wypisują czerwony komunikat na ekran i pozwalają skryptowi jechać dalej, dokładnie tak jak bash bez `set -e` leci dalej po niepowodzeniu komendy. Zdecydowana większość błędów zwracanych przez wbudowane cmdlety (na przykład `Get-Service` na nieistniejącej usłudze) jest domyślnie nieprzerywająca. `$ErrorActionPreference = "Stop"` zmienia to zachowanie globalnie dla bieżącej sesji albo skryptu, zamieniając każdy nieprzerywający błąd w przerywający — bez tego `try/catch` opisany niżej w ogóle nie przechwyci większości błędów cmdletów, bo `catch` reaguje wyłącznie na błędy przerywające.

□□ TYLKO ODCZYT

Wymuszenie błędu przerywającego dla pojedynczego polecenia: -ErrorAction Stop

```
Get-Service -Name NazwaUslugi -ErrorAction Stop
```

Zamiast zmieniać zachowanie całej sesji przez `$ErrorActionPreference`, wymuszamy błąd przerywający tylko dla jednego, konkretnego wywołania — bardziej precyzyjne podejście, gdy reszta skryptu ma normalnie tolerować drobne, nieprzerywające błędy innych poleceń.

□□ TYLKO ODCZYT

Obsługa błędów: try / catch / finally

```
try {
    Get-Service -Name NazwaUslugi -ErrorAction Stop
}
catch {
    Write-Error "Nie udało się pobrać usługi: $($_.Exception.Message)"
}
```

```
finally {
    Write-Output "Sekcja zakończona, niezależnie od wyniku."
}
```

`try` obejmuje kod, który może zawieść, `catch` przechwytuje wyłącznie błędy przerywające (stąd `-ErrorAction Stop` w bloku `try` wyżej — bez niego `catch` nigdy by się nie uruchomił dla tego konkretnego polecenia), a `finally` wykonuje się zawsze, niezależnie od tego, czy `try` się powiodło, czy wpadło w `catch` — dokładnie ta sama gwarancja sprzątania, co w skrypcie zmieniającym plik `hosts` na stronie WinRM: zarządzanie zdalne.

WinRM = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting

TYLKO ODCZYT

Przechwycenie konkretnego typu wyjątku

```
try {
    Get-ADUser -Identity "NazwaUzytkownika" -ErrorAction Stop
}
catch [Microsoft.ActiveDirectory.Management.ADIdentityNotFoundException] {
    Write-Warning "Użytkownik nie istnieje w katalogu."
}
catch {
    Write-Error "Nieoczekiwany błąd: $($_.Exception.Message)"
}
```

Rozróżniamy konkretny, spodziewany typ wyjątku od wszystkich pozostałych — pierwszy pasujący blok `catch` wygrywa, więc kolejność od najbardziej szczegółowego do ogólnego ma znaczenie.

ZMIENIA SYSTEM

Sprawdzenie historii błędów bieżącej sesji: $Error

```
$Error[0] | Format-List * -Force
```

> **Pojęcie: $Error.** Automatyczna tablica przechowująca wszystkie błędy, jakie wystąpiły w bieżącej sesji, od najnowszego (`$Error[0]`) do najstarszego — przydatna

przy diagnozowaniu tuż po awarii, zanim sesja się zamknie, bez konieczności odtwarzania błędu jeszcze raz.

□□ TYLKO ODCZYT

Wygenerowanie własnego błędu: throw

```
if (-not (Test-Path $SciezkaDoPliku)) {
    throw "Plik konfiguracyjny nie istnieje: $SciezkaDoPliku"
}
```

`throw` generuje błąd przerywający z własnym komunikatem — zatrzymuje bieżący zakres wykonania natychmiast, chyba że stoi wewnątrz `try`, wtedy trafia do odpowiadającego `catch`.

□□ TYLKO ODCZYT

Strumienie wyjścia: Write-Output, Write-Host, Write-Verbose, Write-Error

Polecenie	Trafia do potoku	Kiedy używamy
`Write-Output`	tak	zwrócenie wyniku dalej w potoku — to, co funkcja albo skrypt faktycznie „zwraca"
`Write-Host`	nie	komunikat wyłącznie na ekran dla operatora, nigdy nie powinien być dalej przetwarzany przez inny skrypt
`Write-Verbose`	nie, chyba że `-Verbose`	szczegóły diagnostyczne, widoczne tylko po jawnym włączeniu trybu gadatliwego
`Write-Warning`	nie	ostrzeżenie, sytuacja niepokojąca, ale niewymagająca przerwania
`Write-Error`	nie (błąd, nie dane)	zgłoszenie błędu nieprzerywającego z poziomu własnego kodu

Przykład różnicy w praktyce:

```
function Test-Wyjscie {
    Write-Output "to trafi do zmiennej"
    Write-Host "to tylko na ekran"
}
$wynik = Test-Wyjscie
$wynik
```

```
to tylko na ekran
to trafi do zmiennej
```

`Write-Host` wypisało się od razu na ekran w trakcie wykonania funkcji, natomiast `$wynik` po przypisaniu zawiera wyłącznie tekst z `Write-Output` — to jest najczęstsza przyczyna błędów w funkcjach, których wynik ktoś próbuje dalej przetwarzać w potoku, a które przez pomyłkę używają `Write-Host` zamiast `Write-Output`.

◫ OSTROŻNIE · NIEODWRACALNE

Zapowiedź akcji bez jej wykonania: -WhatIf

```
function Remove-StareLogi {
    [CmdletBinding(SupportsShouldProcess)]
    param(
        [string]$Katalog
    )
    Get-ChildItem $Katalog -Filter *.log | ForEach-Object {
        if ($PSCmdlet.ShouldProcess($_.FullName, "Usunięcie pliku")) {
            Remove-Item $_.FullName
        }
    }
}

Remove-StareLogi -Katalog C:\Logs -WhatIf
```

> **Pojęcie: SupportsShouldProcess.** Atrybut `[CmdletBinding(SupportsShouldProcess)]` dodaje funkcji wbudowane wsparcie dla `-WhatIf` (pokaż, co by się stało, nic nie rób) i `-Confirm` (zapytaj przed każdą operacją) — wewnątrz funkcji `$PSCmdlet.ShouldProcess()` sam sprawdza, czy któryś z tych parametrów jest aktywny, i zwraca `$false` zamiast wykonać właściwą akcję, jeśli tak.

Przykład:

```
What if: Performing the operation "Usunięcie pliku" on target "C:\Logs\stary.log".
What if: Performing the operation "Usunięcie pliku" on target "C:\Logs\archiwalny.log".
```

⚠ OSTROŻNIE · NIEODWRACALNE

Wymuszenie potwierdzenia przed każdą operacją: -Confirm

```
Remove-StareLogi -Katalog C:\Logs -Confirm
```

Ta sama funkcja z `-Confirm` pyta o potwierdzenie osobno dla każdego pliku, zanim go usunie — przydatne przy pierwszym uruchomieniu nowego skryptu na produkcji, gdy `-WhatIf` już potwierdziło, że lista celów wygląda poprawnie, ale wciąż chcemy kontrolować tempo wykonania ręcznie.

⚠ ZMIENIA SYSTEM

Rozpoczęcie zapisu całej sesji do pliku: Start-Transcript

```
Start-Transcript -Path "C:\Logi\skrypt-$(Get-Date -Format 'yyyyMMdd-HHmmss').log"
```

> **Pojęcie: transkrypcja.** `Start-Transcript` zapisuje do pliku dosłownie wszystko, co pojawia się w konsoli — wpisywane polecenia i cały ich wynik, łącznie z `Write-Host` — aż do wywołania `Stop-Transcript`. To inny mechanizm niż logowanie własnymi `Write-Output` / `Add-Content`, bo nie wymaga świadomego dopisywania każdej linijki logu ręcznie, kosztem mniejszej kontroli nad formatem zapisanego pliku.

⚠ ZMIENIA SYSTEM

Zapisanie własnego logu do pliku

```
function Write-Log {
    param([string]$Wiadomosc, [string]$Poziom = "INFO")
    $wpis = "{0} [{1}] {2}" -f (Get-Date -Format "yyyy-MM-dd HH:mm:ss"), $Poziom,
$Wiadomosc
    Add-Content -Path "C:\Logi\skrypt.log" -Value $wpis
```

```
}


Write-Log "Rozpoczęto przetwarzanie" -Poziom "INFO"
```

Budujemy własny format logu, gdy potrzebujemy większej kontroli niż daje `Start-Transcript` — jednolity znacznik czasu, poziom ważności i wiadomość w jednej linii, łatwe do dalszego przeszukiwania narzędziem typu `Select-String`.

> **Uważaj**
>
> - `catch` bez `-ErrorAction Stop` na poprzedzającym poleceniu nigdy się nie uruchomi dla większości cmdletów, bo ich domyślne błędy są nieprzerywające.
> - `Write-Host` w funkcji, której wynik ktoś przekazuje dalej w potoku, jest jednym z najczęstszych źródeł „zniknięcia" danych — trafiają na ekran, nie do zmiennej.
> - `Set-StrictMode -Version Latest` na starym, nigdy wcześniej pod nim nie testowanym skrypcie potrafi ujawnić od razu kilkanaście utajonych literówek — warto włączać go od początku pisania nowego kodu, nie doklejać na koniec do gotowego skryptu bez planu na przetestowanie każdego wyjątku.

Z życia wzięte

> **Z ŻYCIA WZIĘTE**
>
> Tomek pisze skrypt restartujący usługę Optimy na sql01 i owija go w `try/catch`, przekonany że jest bezpiecznie. W środę usługa faktycznie nie istnieje pod tą nazwą po literówce w parametrze, `Get-Service` wypisuje czerwony błąd na ekranie, a `catch` mimo to się nie uruchamia. Dopiero dopisanie `-ErrorAction Stop` do `Get-Service` sprawia, że ten sam błąd trafia tam, gdzie powinien.

Po co to istnieje

Skrypt napisany „na szybko" do jednorazowego użycia i skrypt, który ma działać samodzielnie z harmonogramu zadań przez kolejne lata, różnią się przede wszystkim tym, ile założeń można w nich bezpiecznie pominąć. Jednorazowy skrypt piszemy i uruchamiamy sami, na oczach, więc błędną literówkę widzimy natychmiast na ekranie. Skrypt produkcyjny uruchomi ktoś inny, o innej porze, bez patrzenia na ekran — musi sam wykryć, że coś poszło nie tak, zatrzymać się w

bezpiecznym miejscu i zostawić ślad w logu tłumaczący, co konkretnie zawiodło.

> **Pojęcie: skrypt, funkcja, moduł.** Skrypt to pojedynczy plik `.ps1`, uruchamiany bezpośrednio z linii poleceń albo z harmonogramu — ma swój własny zakres zmiennych i kończy działanie po wykonaniu ostatniej linii. Funkcja to nazwany blok kodu wewnątrz skryptu (albo modułu), wywoływany wielokrotnie z różnymi argumentami — istnieje po to, żeby nie powtarzać tego samego fragmentu kodu w kilku miejscach. Moduł to zbiór funkcji, zmiennych i aliasów spakowanych razem i importowanych poleceniem `Import-Module` do dowolnej sesji, która ich potrzebuje — to jest jednostka dystrybucji i ponownego użycia kodu między różnymi skryptami i różnymi osobami, w odróżnieniu od funkcji, która żyje w obrębie jednego pliku, dopóki jej świadomie nie wydzielimy do modułu.

Jak to działa

`Set-StrictMode` i `$ErrorActionPreference` odpowiadają na dwa różne pytania, mylone ze sobą równie często jak `set -e` i `set -u` w bashu. `Set-StrictMode` pilnuje poprawności samego kodu — nieistniejące zmienne, nieistniejące właściwości, złe wywołania metod — niezależnie od tego, czy jakiekolwiek polecenie w ogóle zwróciło błąd. `$ErrorActionPreference` decyduje, co się dzieje, gdy polecenie (cmdlet, funkcja, wywołanie zewnętrznego programu) faktycznie zgłosi błąd — czy skrypt jedzie dalej, czy się zatrzymuje. Oba razem dają pokrycie zbliżone do bashowego `set -euo pipefail`, ale są dwoma osobnymi ustawieniami, które trzeba włączyć osobno, i o czym łatwo zapomnieć, testując skrypt wyłącznie na przypadku, gdzie wszystko się powiodło.

`try/catch` bez `-ErrorAction Stop` na krytycznym poleceniu to pułapka analogiczna do czterech sytuacji, w których bashowe `set -e` nie działa, opisanych na stronie o trybie ścisłym basha — mechanizm istnieje i wygląda na aktywny, ale w praktyce nie łapie niczego, bo błąd, który miał go uruchomić, nigdy nie stał się przerywający.



Bez -ErrorAction Stop blad nieprzerywajacy omija catch, z nim trafia do niego poprawnie (kliknięcie otwiera pełny rozmiar)

Częste błędy

Poleganie na samym `try/catch` bez `$ErrorActionPreference` albo `-ErrorAction Stop`. Skrypt „przechodzi" przez `catch` bez wejścia do niego, mimo widocznego czerwonego komunikatu błędu na ekranie chwilę wcześniej — bo ten konkretny błąd był nieprzerywający i `try/catch` nigdy go nie zobaczył.

Zwracanie wyniku funkcji przez Write-Host zamiast Write-Output. Kod działa poprawnie przy ręcznym uruchomieniu (bo widać wynik na ekranie), a przestaje działać, gdy ktoś spróbuje przypisać go do zmiennej albo puścić dalej w potoku — `$wynik = Moja-Funkcja` zwraca `$null` albo pusty ciąg, mimo że na ekranie coś się wyświetliło.

Brak `-WhatIf` przy pierwszym uruchomieniu skryptu usuwającego dane na produkcji. `SupportsShouldProcess` istnieje właśnie po to, żeby dało się bezpiecznie zweryfikować listę celów przed wykonaniem nieodwracalnej operacji — pominięcie tego kroku i uruchomienie od razu bez `-WhatIf` jest samo w sobie błędem proceduralnym, niezależnie od tego, czy skrypt akurat zawiera błąd w logice.

Szkielet skryptu produkcyjnego

```
#Requires -Version 5.1
<#
.SYNOPSIS
    Krótki opis tego, co robi skrypt.
.DESCRIPTION
    Dłuższy opis, parametry, przykład wywołania.
.EXAMPLE
    .\SzkieletSkryptu.ps1 -NazwaSerwera wsus01 -Akcja Restart
#>

[CmdletBinding(SupportsShouldProcess)]
param(
    [Parameter(Mandatory)]
    [string]$NazwaSerwera,

    [ValidateSet("Restart", "Stop", "Start")]
```

```
    [string]$Akcja = "Start",

    [string]$SciezkaLogu = "C:\Logi\$($MyInvocation.MyCommand.Name)-$(Get-Date -Format
'yyyyMMdd').log"
)

Set-StrictMode -Version Latest
$ErrorActionPreference = "Stop"

function Write-Log {
    param([string]$Wiadomosc, [string]$Poziom = "INFO")
    $wpis = "{0} [{1}] {2}" -f (Get-Date -Format "yyyy-MM-dd HH:mm:ss"), $Poziom, $Wiadomosc
    Add-Content -Path $SciezkaLogu -Value $wpis
    Write-Verbose $wpis
}

try {
    Write-Log "Rozpoczęto: akcja=$Akcja, serwer=$NazwaSerwera"

    if ($PSCmdlet.ShouldProcess($NazwaSerwera, "Wykonanie akcji $Akcja")) {
        switch ($Akcja) {
            "Restart" { Restart-Service -InputObject (Get-Service -ComputerName $NazwaSerwera
-Name NazwaUslugi) }
            "Stop"    { Stop-Service -InputObject (Get-Service -ComputerName $NazwaSerwera -
Name NazwaUslugi) }
            "Start"   { Start-Service -InputObject (Get-Service -ComputerName $NazwaSerwera -
Name NazwaUslugi) }
        }
    }

    Write-Log "Zakończono pomyślnie"
}
catch {
    Write-Log "BŁĄD: $($_.Exception.Message)" -Poziom "ERROR"
    throw
}
finally {
    Write-Log "Koniec wykonania skryptu"
}
```

Ten szkielet łączy wszystkie elementy z tej strony naraz: nagłówek `.SYNOPSIS`/`.DESCRIPTION` czytany przez `Get-Help`, parametry z walidacją, tryb ścisły, jednolite logowanie, wsparcie `-WhatIf`/`-Confirm` i `try/catch/finally` z ponownym rzuceniem błędu (`throw` bez argumentu w bloku `catch` przekazuje oryginalny wyjątek dalej, zamiast go tłumić) po zapisaniu do logu.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell, w katalogu `$env:TEMP`, bez uprawnień administratora.

Ćwiczenie 1 — Write-Host kontra Write-Output

```
function Test-Funkcja1 { Write-Host "wynik A" }
function Test-Funkcja2 { Write-Output "wynik B" }

$a = Test-Funkcja1
$b = Test-Funkcja2

"a = [$a]"
"b = [$b]"
```

Przewidujemy, co znajdzie się w `$a` i `$b`, zanim uruchomimy kod.

Odpowiedź

`$a` będzie puste (`a = []`), bo `Write-Host` wypisało tekst bezpośrednio na ekran i nigdy nie trafiło do potoku, więc przypisanie do zmiennej nie miało czego przechwycić. `$b` będzie zawierać `wynik B`, bo `Write-Output` przekazuje wartość dalej w potoku, skąd przypisanie do zmiennej ją odbiera. To jest praktyczna demonstracja tabeli strumieni wyjścia z fiszki wyżej.

Ćwiczenie 2 — tryb ścisły ujawnia literówkę

```
function Sprawdz-Usluge {
    param($Nazwa)
    $usluga = Get-Service -Name $Nazwa
    if ($usluga.StatuS -eq "Running") {
        "Usługa działa"
    } else {
        "Usługa nie działa"
    }
}

Sprawdz-Usluge -Nazwa "Spooler"

Set-StrictMode -Version Latest
Sprawdz-Usluge -Nazwa "Spooler"
```

Zauważamy literówkę `StatuS` zamiast `Status` w kodzie funkcji — sprawdzamy, jak zachowuje się bez i z trybem ścisłym.

Odpowiedź

Bez `Set-StrictMode` odwołanie do nieistniejącej właściwości `StatuS` po cichu zwraca `$null`, porównanie `$null -eq "Running"` daje `$false`, więc funkcja zawsze zgłasza „Usługa nie działa" — nawet jeśli usługa faktycznie działa. Po włączeniu `Set-StrictMode -Version Latest` to samo wywołanie rzuca jawny błąd `PropertyNotFoundException`, natychmiast pokazując literówkę zamiast cichej, błędnej odpowiedzi. To jest dokładnie ten sam mechanizm, co `set -u` w bashu wyłapujące niezdefiniowaną zmienną.

Pytania kontrolne

1. Dlaczego `try { Get-Service NieistniejacaUsluga } catch { "złapano" }` nie wypisuje „złapano", mimo że usługa naprawdę nie istnieje?

Odpowiedź

Domyślny błąd `Get-Service` dla nieistniejącej usługi jest nieprzerywający — wypisuje czerwony komunikat na ekran i pozwala wykonaniu jechać dalej, więc blok `catch` nigdy się nie aktywuje, bo nie doszło do błędu przerywającego. Naprawą jest dodanie `-ErrorAction Stop` do wywołania `Get-Service` wewnątrz `try`, dokładnie jak w fiszce „Obsługa błędów" wyżej.

2. Funkcja ma `[CmdletBinding(SupportsShouldProcess)]`, ale w jej treści nigdzie nie pojawia się `$PSCmdlet.ShouldProcess()`. Co się stanie po wywołaniu jej z `-WhatIf`?

Odpowiedź

Sam atrybut `SupportsShouldProcess` tylko udostępnia parametry `-WhatIf` i `-Confirm` na poziomie wywołania funkcji — nie robi nic automatycznie z samą logiką wewnątrz. Bez jawnego wywołania `$PSCmdlet.ShouldProcess()` w miejscu, gdzie faktycznie wykonuje się zmieniająca stan operacja, funkcja przyjmie `-WhatIf` bez błędu składni, ale i tak wykona wszystkie operacje normalnie — `-WhatIf` bez odpowiadającego mu `ShouldProcess()` w kodzie jest atrapą, nie realną ochroną.

Powiązane: Tryb ścisły: set -euo pipefail · Podstawy składni PowerShella · Polityka wykonywania i podpisywanie skryptów · Poświadczenia i bezpieczne hasła w skryptach · Wysyłanie poczty ze skryptu

Źródła: `Get-Help about_Functions_Advanced` · `Get-Help about_Try_Catch_Finally` · `Get-Help about_CommonParameters` · dokumentacja Microsoft Learn, „about Preference Variables"

Zadania zaplanowane, usługi i procesy

Poziom: średniozaawansowany **Dotyczy: Windows 10/11, Windows Server 2016+**

Czas: ~30 min

Trzy warstwy tego, co dzieje się na serwerze bez naszego udziału — zaplanowane zadania, usługi działające w tle i procesy — oraz dziennik zdarzeń jako jedyne wiarygodne źródło odpowiedzi na pytanie „kto to zrobił i kiedy".

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
CIM	Common Information Model: standard opisu obiektów zarządzania w Windows, na nim stoi WMI
gMSA	group Managed Service Account: konto usługi, któremu Windows sam zmienia hasło
WMI	Windows Management Instrumentation: interfejs do odczytu stanu Windowsa
WMS	Warehouse Management System: system zarządzania magazynem

Fiszka

ZMIENIA SYSTEM

Wyszukanie zadania w Task Scheduler o odpowiedniej nazwie

```
# Definiujemy nazwę zadania do wyszukania
$taskName = 'NazwaZadania'

# Pobierz nazwę hosta i skonstruuj nazwę pliku
$hostname = $env:COMPUTERNAME
$filepath = "tasks_$hostname.txt"
```

```
# Wypisz wszystkie zadania wraz z nazwami i ścieżkami, zapisz do pliku
Get-ScheduledTask | Select-Object TaskName, TaskPath | Out-File -FilePath $filepath -
Encoding UTF8

# Oddziel sekcje w pliku
Add-Content -Path $filepath "`nWeryfikacja czy istnieje task o określonej nazwie:`n"

# Sprawdź, czy zadanie o podanej nazwie istnieje, dopisz wynik na końcu pliku lub
stosowny komunikat
$task = Get-ScheduledTask | Where-Object { $_.TaskName -eq $taskName }
if ($task) {
    $task | Out-File -FilePath $filepath -Append -Encoding UTF8
} else {
    Add-Content -Path $filepath "Brak zadania o nazwie $taskName"
}
```

Skrypt uruchamiamy w PowerShell z uprawnieniami Administratora. Zapisuje pliki w aktualnie używanym folderze — przechodzimy najpierw do katalogu docelowego, zanim go uruchomimy. Przydatny, gdy trzeba potwierdzić na wielu serwerach naraz (przez `Invoke-Command` opisane na stronie WinRM: zarządzanie zdalne), czy dane zadanie wdrożeniowe faktycznie istnieje pod oczekiwaną nazwą, zamiast otwierać konsolę Harmonogramu zadań na każdej maszynie z osobna.

Przykład:

```
Weryfikacja czy istnieje task o określonej nazwie:

TaskPath      : \
TaskName      : NazwaZadania
State         : Ready
```

□□ **WinRM** = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting · **hostname** = Hostname: wypisuje nazwę komputera

□□ TYLKO ODCZYT

Wylistowanie wszystkich zadań zaplanowanych: Get-ScheduledTask

```
Get-ScheduledTask | Where-Object State -ne "Disabled" | Format-Table TaskName, TaskPath,
State -AutoSize
```

Zwracamy wszystkie aktywne zadania (pomijając wyłączone) wraz z ich stanem — `Ready` oznacza gotowość do uruchomienia zgodnie z wyzwalaczem, `Running` oznacza trwające właśnie wykonanie.

Przykład:

```
TaskName                TaskPath    State
--------                --------    -----
GoogleUpdateTaskMachine \           Ready
BackupCodzienny         \Firma\     Running
```

◫◫ ZMIENIA SYSTEM

Rejestracja nowego zadania zaplanowanego: Register-ScheduledTask

```
$akcja = New-ScheduledTaskAction -Execute "PowerShell.exe" -Argument "-File
SciezkaDoPliku"
$wyzwalacz = New-ScheduledTaskTrigger -Daily -At "03:00"
$ustawienia = New-ScheduledTaskSettingsSet -StartWhenAvailable

Register-ScheduledTask -TaskName "NazwaZadania" -Action $akcja -Trigger $wyzwalacz -
Settings $ustawienia -User "SYSTEM"
```

Budujemy zadanie z trzech osobnych obiektów — akcji (co uruchomić), wyzwalacza (kiedy) i ustawień (jak się zachować, np. przy braku zasilania albo spóźnionym starcie) — i rejestrujemy je jednym poleceniem. `-User "SYSTEM"` uruchamia zadanie na koncie systemowym, bez potrzeby przechowywania hasła — dla zadań wymagających konta z hasłem odsyłamy do strony Poświadczenia i bezpieczne hasła w skryptach.

◫◫ ZMIENIA SYSTEM

Ręczne uruchomienie zadania: Start-ScheduledTask

```
Start-ScheduledTask -TaskName "NazwaZadania"
```

Uruchamiamy zadanie natychmiast, z pominięciem jego wyzwalacza — przydatne przy testowaniu nowo zarejestrowanego zadania bez czekania na najbliższą zaplanowaną porę.

🔒 TYLKO ODCZYT

Sprawdzenie historii wykonań zadania

```
Get-ScheduledTaskInfo -TaskName "NazwaZadania" | Select-Object LastRunTime,
LastTaskResult, NextRunTime
```

> **Pojęcie: LastTaskResult.** Kod zakończenia ostatniego uruchomienia zadania — `0` oznacza sukces, każda inna wartość to kod błędu zwrócony przez uruchomiony program albo skrypt. Sam `Get-ScheduledTask` nie pokazuje tej informacji — trzeba dociągnąć ją osobnym poleceniem `Get-ScheduledTaskInfo`, analogicznie do dociągania nazwy procesu do połączenia sieciowego na stronie Sieć z wiersza poleceń.

Przykład:

```
LastRunTime      LastTaskResult NextRunTime
-----------      -------------- -----------
24.09.2026 03:00              0 25.09.2026 03:00
```

🔒 TYLKO ODCZYT

Pełna historia zdarzeń zadania z dziennika systemowego

```
Get-WinEvent -LogName "Microsoft-Windows-TaskScheduler/Operational" -FilterXPath
"*[EventData[Data[@Name='TaskName']='\NazwaZadania']]"
```

`Get-ScheduledTaskInfo` pokazuje tylko ostatnie uruchomienie — pełną historię z konkretnymi znacznikami czasu każdego startu, zakończenia i ewentualnego błędu trzeba wyciągnąć z dziennika operacyjnego harmonogramu zadań, który domyślnie bywa wyłączony i wymaga włączenia w Podglądzie zdarzeń (`Applications and Services Logs\Microsoft\Windows\TaskScheduler\Operational`).

🔒 TYLKO ODCZYT

Sprawdzenie stanu usługi: Get-Service

```
Get-Service -Name NazwaUslugi
```

Przykład:

```
Status   Name               DisplayName
------   ----               -----------
Running  wuauserv           Windows Update
```

⚠️ ZMIENIA SYSTEM

Zmiana typu uruchomienia usługi: Set-Service

```
Set-Service -Name NazwaUslugi -StartupType Automatic
```

Zmieniamy, czy usługa startuje automatycznie z systemem, ręcznie na żądanie, czy jest wyłączona (`Automatic`, `Manual`, `Disabled`) — bez wpływu na jej bieżący stan działania, jeśli akurat już działa albo stoi.

⚠️ ZMIENIA SYSTEM

Zrestartowanie usługi: Restart-Service

```
Restart-Service -Name NazwaUslugi -Force
```

`-Force` restartuje też usługi zależne, które musiałyby zostać zatrzymane, żeby zatrzymać usługę główną — bez tego parametru polecenie odmawia restartu, jeśli inne usługi deklarują zależność od tej wskazanej.

Przykład błędu bez `-Force`:

```
Restart-Service : Cannot stop service 'NazwaUslugi' because it has dependent services.
```

✅ TYLKO ODCZYT

Sprawdzenie konta, na którym działa usługa: Get-CimInstance Win32_Service

```
Get-CimInstance -ClassName Win32_Service -Filter "Name='NazwaUslugi'" | Select-Object Name, StartName, State
```

> **Pojęcie: Win32_Service.** `Get-Service` nie pokazuje, na jakim koncie faktycznie działa usługa — tę informację (właściwość `StartName`) trzeba pobrać przez WMI/CIM, osobny podsystem zarządzania niż ten, na którym operuje `Get-Service`. Przydatne przy audycie, które usługi działają na koncie `LocalSystem`, a które na dedykowanym koncie usługi wymagającym rotacji hasła — albo na koncie gMSA, opisanym na stronie Konfiguracja kont MSA / gMSA.

Przykład:

```
Name        StartName             State
----        ---------              -----
MSSQLSERVER NT SERVICE\MSSQLSERVER Running
```

WMI = Windows Management Instrumentation: interfejs do odczytu stanu Windowsa · **gMSA** = group Managed Service Account: konto usługi, któremu Windows sam zmienia hasło · **MSA** = Managed Service Account: konto usługi z automatyczną zmianą hasła, dla jednego serwera

TYLKO ODCZYT

Wylistowanie procesów z zużyciem zasobów: Get-Process

```
Get-Process | Sort-Object WorkingSet64 -Descending | Select-Object -First 5 ProcessName, Id, CPU, WorkingSet64
```

Przykład:

```
ProcessName    Id        CPU  WorkingSet64
-----------    --        ---  ------------
sqlservr     5560     892.44     942813184
chrome       8420     412.34     487194112
```

CPU = Central Processing Unit: procesor

ZMIENIA SYSTEM

Zakończenie procesu: Stop-Process

```
Stop-Process -Id NumerPID -Force
```

`-Force` kończy proces bez pytania o potwierdzenie — przydatne w skrypcie automatycznym, ale warto najpierw sprawdzić `-WhatIf`, opisane na stronie Skrypty PowerShell: struktura i dobre praktyki, zanim ubijemy proces, który miał zostać przy życiu.

🔒 TYLKO ODCZYT

Znalezienie procesu trzymającego dany port

```
$port = NumerPortu
$pid = (Get-NetTCPConnection -LocalPort $port -ErrorAction
SilentlyContinue).OwningProcess
Get-Process -Id $pid
```

Łączymy `Get-NetTCPConnection` opisane na stronie Sieć z wiersza poleceń z `Get-Process`, żeby jednym poleceniem odpowiedzieć na pytanie „co dokładnie zajmuje ten port" — częste przy diagnozowaniu błędu `port already in use` zaraz po próbie startu nowej usługi.

Przykład:

```
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
   1204      42   88320      94512      12.34   4488   0 w3wp
```

🔒 TYLKO ODCZYT

Odpytanie dziennika zdarzeń filtrem: Get-WinEvent -FilterHashtable

```
Get-WinEvent -FilterHashtable @{LogName = 'System'; Id = 1074} | Format-Table -Property
TimeCreated, Message -Wrap
```

> **Pojęcie: Get-WinEvent kontra Get-EventLog.** `Get-EventLog` to starsze polecenie, ograniczone do klasycznych dzienników (`Application`, `System`, `Security`) i wolniejsze przy dużych dziennikach, bo odczytuje zdarzenia sekwencyjnie i filtruje je już po stronie klienta PowerShella. `Get-WinEvent` z `-FilterHashtable` przekazuje filtr (nazwę dziennika, identyfikator zdarzenia, zakres czasu) bezpośrednio do silnika ETW/EventLog systemu Windows, który filtruje zdarzenia przed ich przesłaniem — przy dziennikach liczonych w dziesiątkach tysięcy wpisów różnica w czasie wykonania bywa rzędu wielkości. `Get-WinEvent` obsługuje też nowsze dzienniki aplikacyjne i usługowe (

`Microsoft-Windows-...`), których `Get-EventLog` w ogóle nie widzi.

Przykład:

```
TimeCreated          Message
-----------          -------
24.09.2026 22:14:02  The process C:\Windows\System32\shutdown.exe (SQL01) has initiated
the restart
                     of computer SQL01 on behalf of user DOMENA\nazwa_usera for the
following
                     reason: Other (Planned)
                     Reason Code: 0x80000000
                     Shutdown Type: restart
                     Comment:
```

TYLKO ODCZYT

Filtrowanie po kilku identyfikatorach zdarzeń naraz

```
Get-WinEvent -FilterHashtable @{LogName = 'System'; Id = 1074, 6006, 6008, 41} |
    Sort-Object TimeCreated -Descending |
    Select-Object TimeCreated, Id, Message
```

Łączymy w jednym zapytaniu identyfikatory pokrywające pełny cykl zamknięcia i startu systemu, opisane w tabeli niżej — jedno polecenie pokazuje chronologię: kto zainicjował zamknięcie, czy dziennik zdarzeń zdążył się zatrzymać poprawnie, i czy poprzedni start nie był nieoczekiwanym odzyskaniem po awarii.

TYLKO ODCZYT

Wszystkie identyfikatory zdarzeń związane z restartem i zamknięciem systemu

ID zdarzenia	Dziennik	Znaczenie
`1074`	System	proces albo użytkownik zainicjował zamknięcie lub restart — zawiera nazwę procesu inicjującego, konto użytkownika i podany powód

ID zdarzenia	Dziennik	Znaczenie
`6006`	System	usługa Dziennika zdarzeń zatrzymała się poprawnie — oznacza czyste zamknięcie systemu
`6008`	System	poprzednie zamknięcie systemu było nieoczekiwane — brak zdarzenia `6006` przed startem oznacza zawieszenie, wyłączenie zasilania albo awarię, nie planowe zamknięcie
`41`	System	system uruchomił się ponownie bez czystego zamknięcia (`Kernel-Power`) — najczęściej twardy reset, zanik zasilania albo BSOD, silniejszy sygnał awarii niż samo `6008`

Uważaj

- Zdarzenie `1074` mówi, kto zainicjował zamknięcie z poziomu systemu operacyjnego — nie zarejestruje twardego wyłączenia zasilania ani resetu sprzętowego, na to wskazuje dopiero `41` albo `6008`.
- `Restart-Service -Force` na usłudze z wieloma zależnymi usługami restartuje też je wszystkie — warto sprawdzić `Get-Service -Name NazwaUslugi -RequiredServices` i `-DependentServices` przed użyciem na usłudze współdzielonej przez wiele innych komponentów.
- `Get-WinEvent -FilterHashtable` z błędną nazwą klucza (na przykład literówka `Level` zamiast `LevelDisplayName`) zwraca cichy błąd walidacji parametru, a nie pusty wynik — czytelny komunikat błędu ułatwia poprawkę, o ile nie jest przechwycony przez zbyt szeroki `try/catch`.

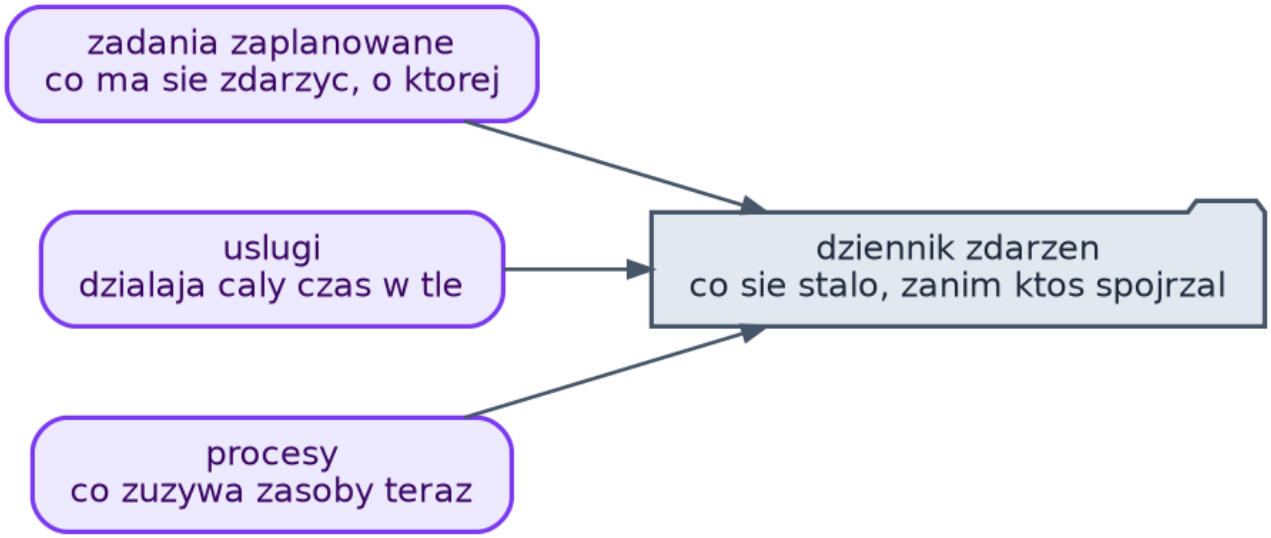
Z życia wzięte

□□ Z ŻYCIA WZIĘTE

Marek zgłasza w poniedziałek o 6:00, że skanery w Swarzędzu nie widzą WMS na ora01, mimo że wszystko wyglądało dobrze w piątek. Sprawdzamy `Get-ScheduledTaskInfo` dla nocnego zadania synchronizacji i `LastTaskResult` pokazuje `1`, nie `0`. Dopiero `Get-WinEvent` na dzienniku operacyjnym harmonogramu pokazuje, że zadanie uruchomiło się o czasie, ale zakończyło błędem zaraz po starcie, bo konto usługowe miało tego dnia wygasłe hasło.

Po co to istnieje

Trzy poziomy tej strony odpowiadają na trzy różne pytania o to, co dzieje się na serwerze bez bieżącego nadzoru administratora. Zadania zaplanowane odpowiadają na „co ma się wydarzyć samo, o określonej porze". Usługi odpowiadają na „co działa non stop w tle, niezależnie od tego, czy ktokolwiek jest zalogowany". Procesy odpowiadają na „co faktycznie zużywa zasoby albo trzyma port w tej właśnie chwili". Dziennik zdarzeń spina wszystkie trzy poziomy wstecz w czasie — jedyne miejsce, gdzie da się sprawdzić, co się stało, zanim ktokolwiek zdążył spojrzeć na ekran.



Trzy warstwy tego, co dzieje sie na serwerze, spiete wspolnym dziennikiem zdarzen (kliknięcie otwiera pełny rozmiar)

Jak to działa

Różnica między `Get-Service` a `Get-CimInstance Win32_Service` wynika z tego, że są to dwa osobne podsystemy zarządzania Windowsem o częściowo pokrywającej się, ale nie identycznej odpowiedzialności. `Get-Service` korzysta z Menedżera kontroli usług (SCM) i zwraca szybko podstawowy stan — działa/nie działa, typ uruchomienia. WMI/CIM ma znacznie szerszy, ale wolniejszy dostęp do metadanych usługi, w tym konta wykonawczego, ścieżki binariów i opisu — sięgamy po CIM dokładnie wtedy, gdy potrzebna informacja nie jest wystawiona przez prostszy, szybszy SCM.

Sekwencja zdarzeń `1074` → `6006` → (restart) → brak `6008` na starcie oznacza planowe, czyste zamknięcie zainicjowane przez kogoś świadomie. Brak `6006` przed `6008` na kolejnym starcie oznacza, że dziennik zdarzeń nie zdążył zapisać swojego własnego, poprawnego zamknięcia — silny sygnał, że system przestał działać nagle, niezależnie od tego, czy `1074` w ogóle wystąpiło wcześniej. `41` z dziennika `Kernel-Power` jest zapisywane przez samo jądro systemu przy starcie po wykryciu, że poprzednie zamknięcie nie przeszło przez normalną sekwencję zamykania — najsilniejszy z czterech sygnałów, bo pochodzi z warstwy niżej niż usługa Dziennika zdarzeń, która przy prawdziwej awarii zasilania też nie zdążyła nic zapisać.

Częste błędy

Wnioskowanie z samego `1074`, że restart był planowy i bezproblemowy. `1074` rejestruje inicjację restartu, nie jego przebieg — system mógł zawiesić się w trakcie samego zamykania, co ujawni dopiero brak `6006` i obecność `6008` na kolejnym starcie.

Użycie `Get-EventLog` na serwerze z dziennikiem `System` liczącym setki tysięcy wpisów. Polecenie działa poprawnie, ale zauważalnie wolniej niż `Get-WinEvent -FilterHashtable` z tym samym filtrem, bo filtruje po stronie klienta zamiast delegować filtrowanie do samego systemu.

Restart usługi bez `-Force` na usłudze z zależnościami i porzucenie próby po pierwszym błędzie. Komunikat `Cannot stop service... because it has dependent services` jest informacją, nie ślepym zaułkiem — `Get-Service -Name NazwaUslugi -DependentServices` pokazuje dokładnie, co jeszcze zostanie zrestartowane razem z usługą główną, zanim doda się `-Force`.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell. Restart usług i zamykanie procesów wymaga ostrożności — testujemy na usługach i procesach, których zatrzymanie nie przeszkodzi w codziennej pracy.

Ćwiczenie 1 — kto trzyma port 3389

```
$pid = (Get-NetTCPConnection -LocalPort 3389 -ErrorAction SilentlyContinue | Select-Object -
First 1).OwningProcess
if ($pid) {
    Get-Process -Id $pid | Select-Object ProcessName, Id, StartTime
} else {
    "Port 3389 nie jest aktualnie zajęty"
```

```
}
```

Sprawdzamy, jaki proces obsługuje port używany przez Pulpit zdalny na naszej maszynie.

Odpowiedź

Na typowej maszynie z Windowsem z włączonym Pulpitem zdalnym port `3389` trzyma proces `svchost` (usługa `TermService` działa wewnątrz współdzielonego procesu hosta usług, nie jako osobny plik wykonywalny) — sama nazwa procesu nie mówi wprost, o którą usługę chodzi, do tego posłużyłoby dodatkowo sprawdzenie `Get-CimInstance Win32_Service -Filter "ProcessId=$pid"`.

Ćwiczenie 2 — sekwencja zdarzeń ostatniego restartu

```
Get-WinEvent -FilterHashtable @{LogName = 'System'; Id = 1074, 6005, 6006, 6008, 41} -
MaxEvents 20 |
    Sort-Object TimeCreated |
    Select-Object TimeCreated, Id, LevelDisplayName
```

Odtwarzamy chronologię ostatniego zamknięcia i startu własnej maszyny na podstawie samych identyfikatorów zdarzeń.

Odpowiedź

Poprawna, planowa sekwencja to `1074` (inicjacja), `6006` (dziennik zatrzymał się poprawnie), a po ponownym starcie `6005` (dziennik wystartował) — bez `6008` ani `41` pomiędzy nimi. Obecność `6008` albo `41` bez poprzedzającego czystego `6006` wskazuje na zamknięcie, które nie przeszło przez normalną sekwencję, niezależnie od tego, czy wcześniej zarejestrowano jakiekolwiek `1074`.

Pytania kontrolne

1. Dlaczego `Get-ScheduledTask` sam z siebie nie pokazuje, czy ostatnie uruchomienie zadania się powiodło?

Odpowiedź

`Get-ScheduledTask` zwraca definicję i bieżący stan zadania (`Ready`, `Running`, `Disabled`), nie historię jego wykonań — informacje o wyniku ostatniego uruchomienia (`LastTaskResult`, `LastRunTime`) przechowuje osobna struktura, dostępna przez `Get-ScheduledTaskInfo`, analogicznie do tego, jak `Get-NetTCPConnection` nie zna nazwy procesu bez osobnego zapytania do `Get-Process`.

2. Serwer zgłasza zdarzenie `41` z dziennika Kernel-Power, ale nie ma żadnego poprzedzającego `1074`. Co to sugeruje?

Odpowiedź

Brak `1074` oznacza, że żaden proces ani użytkownik nie zainicjował zamknięcia z poziomu systemu operacyjnego w sposób, który Windows zdążyłby zarejestrować — w połączeniu z `41` (restart bez czystego zamknięcia) to silny sygnał zewnętrznej przyczyny: zaniku zasilania, twardego resetu sprzętowego albo awarii na poziomie niższym niż system operacyjny, a nie planowego działania administratora czy użytkownika.

Powiązane: Sieć z wiersza poleceń · Skrypty PowerShell: struktura i dobre praktyki · Konfiguracja kont MSA / gMSA · Wysyłanie poczty ze skryptu

Źródła: `Get-Help Get-WinEvent` · `Get-Help Get-ScheduledTask` · dokumentacja Microsoft Learn, „Event ID 41 Kernel-Power" i „Task Scheduler cmdlets"

Wysyłanie poczty ze skryptu

Poziom: średniozaawansowany **Dotyczy: Windows PowerShell 5.1, PowerShell 7+**

Czas: ~20 min

`Send-MailMessage` *jest oznaczone przez Microsoft jako przestarzałe od kilku lat, ale wciąż działa i wciąż jest tym, co większość istniejących skryptów administracyjnych realnie wywołuje — ta strona pokazuje, jak go używać świadomie, i czym warto je docelowo zastąpić.*

Skróty użyte na tej stronie (9)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
HTML	HyperText Markup Language: format tekstowy stron WWW, tu używany jako format raportu GPO do przeglądania w przeglądarce
IT	Information Technology: dział albo praca związana z komputerami i infrastrukturą firmy
OAuth	Open Authorization: standard logowania jednej usługi do drugiej bez podawania jej hasła, przez token wydany po zalogowaniu w przeglądarce
SMTP	Simple Mail Transfer Protocol: protokół wysyłania poczty
SSL	Secure Sockets Layer: dawna nazwa szyfrowania, dziś formalnie TLS
STARTTLS	włączenie szyfrowania w trakcie już nawiązanej, nieszyfrowanej sesji pocztowej
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

Fiszka

ZMIENIA SYSTEM

Wysłanie prostej wiadomości e-mail: Send-MailMessage

```
$EmailFrom = "AdresNadawcy@NazwaDomeny"
$EmailTo = "AdresOdbiorcy@NazwaDomeny"
```

```
$SMTPServer = "AdresSerwera"
$SMTPPort = 587
$SMTPUsername = "NazwaKontaSMTP"
$SMTPPassword = "HasloDoSkrzynki"
$Subject = "TematWiadomosci"
$Body = "TrescWiadomosci"

$SMTPPasswordSecure = ConvertTo-SecureString $SMTPPassword -AsPlainText -Force
$Credential = New-Object System.Management.Automation.PSCredential ($SMTPUsername,
$SMTPPasswordSecure)

Send-MailMessage -From $EmailFrom -To $EmailTo -Subject $Subject -Body $Body -SmtpServer
$SMTPServer -Port $SMTPPort -Credential $Credential -UseSsl
```

Uruchamiamy skrypt z poziomu PowerShella. Budujemy poświadczenia do skrzynki SMTP tym samym mechanizmem `PSCredential`, co opisany na stronie Poświadczenia i bezpieczne hasła w skryptach — hasło jawnym tekstem w zmiennej `$SMTPPassword` jest tu pokazane dla czytelności przykładu, w skrypcie produkcyjnym powinno pochodzić z zaszyfrowanego pliku albo z magazynu `SecretStore`, nie z linijki kodu.

Przykład:

```
PS C:\Skrypty> .\wyslij-mail.ps1
PS C:\Skrypty>
```

`Send-MailMessage` przy powodzeniu nie wypisuje niczego na ekran — brak komunikatu jest sygnałem sukcesu, błąd wysyłki zawsze zwraca czerwony wyjątek.

SMTP = Simple Mail Transfer Protocol: protokół wysyłania poczty

ZMIENIA SYSTEM

Wysłanie wiadomości z treścią HTML

```
$Body = "<h2>Raport dzienny</h2><p>Wszystkie zadania zakończone <b>poprawnie</b>.</p>"

Send-MailMessage -From $EmailFrom -To $EmailTo -Subject $Subject -Body $Body -BodyAsHtml
-SmtpServer $SMTPServer -Port $SMTPPort -Credential $Credential -UseSsl
```

`-BodyAsHtml` mówi klientowi pocztowemu odbiorcy, żeby zinterpretował treść jako znaczniki HTML zamiast wyświetlić je jako zwykły tekst — przydatne przy automatycznych raportach z tabelami i pogrubieniami zamiast surowego tekstu.

HTML = HyperText Markup Language: format tekstowy stron WWW, tu używany jako format raportu GPO do przeglądania w przeglądarce

TYLKO ODCZYT

Wysłanie wiadomości z załącznikiem

```
Send-MailMessage -From $EmailFrom -To $EmailTo -Subject $Subject -Body $Body `
    -Attachments "SciezkaDoPliku" -SmtpServer $SMTPServer -Port $SMTPPort -Credential
$Credential -UseSsl
```

`-Attachments` przyjmuje jedną ścieżkę albo tablicę kilku ścieżek naraz (`-Attachments "plik1.csv", "plik2.log"`) — typowe zastosowanie to dołączenie raportu CSV wygenerowanego wcześniej w tym samym skrypcie, na przykład tego z fiszki Zapisanie listy połączeń do pliku CSV ze strony o sieci.

CSV = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami

TYLKO ODCZYT

Wysłanie wiadomości do wielu odbiorców naraz

```
$odbiorcy = @("AdresOdbiorcy1@NazwaDomeny", "AdresOdbiorcy2@NazwaDomeny")

Send-MailMessage -From $EmailFrom -To $odbiorcy -Cc "AdresDoWiadomosci@NazwaDomeny" -
Subject $Subject -Body $Body -SmtpServer $SMTPServer -Port $SMTPPort -Credential
$Credential -UseSsl
```

`-To` przyjmuje tablicę adresów tak samo jak pojedynczy tekst — `-Cc` i `-Bcc` działają analogicznie, każdy jako osobny parametr.

> **Uważaj**

- Microsoft oznaczył `Send-MailMessage` jako przestarzałe (`deprecated`), bo polecenie nie wspiera nowoczesnych mechanizmów uwierzytelniania (OAuth 2.0, wymagane dziś przez Microsoft 365 i Gmail) i opiera się na starszej bibliotece .NET (`System.Net.Mail`), której Microsoft sam już nie rozwija ani nie łata pod kątem nowych standardów bezpieczeństwa transportu.
- Uwierzytelnianie logujące się samym loginem i hasłem (bez OAuth) do skrzynki Microsoft 365 przestaje działać wraz z wygaszaniem uwierzytelniania podstawowego (Basic Authentication) przez Microsoft — `Send-MailMessage` w tym scenariuszu wymaga konta SMTP z wyjątkiem albo przekierowania przez lokalny przekaźnik SMTP.
- Wiadomość bez `-UseSsl` na porcie 587 zwykle zawiedzie z błędem serwera odmawiającego uwierzytelnienia w połączeniu jawnym — `-UseSsl` uruchamia STARTTLS, opisane niżej.

OAuth = Open Authorization: standard logowania jednej usługi do drugiej bez podawania jej hasła, przez token wydany po zalogowaniu w przeglądarce · **SMTP** = Simple Mail Transfer Protocol: protokół wysyłania poczty · **STARTTLS** = włączenie szyfrowania w trakcie już nawiązanej, nieszyfrowanej sesji pocztowej

Z życia wzięte

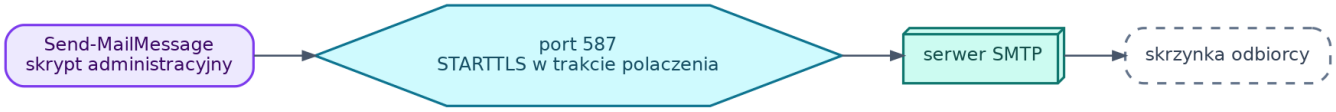
Z ŻYCIA WZIĘTE

My, zespół IT, mamy od dawna skrypt na wsus01 wysyłający co noc raport z zainstalowanych aktualizacji przez `Send-MailMessage` na skrzynkę Microsoft 365. Pewnego dnia raporty przestają przychodzić, a skrypt zwraca błąd uwierzytelnienia, mimo że hasło konta się nie zmieniło. Okazuje się, że dział bezpieczeństwa Microsoftu wygasił uwierzytelnianie podstawowe dla tej dzierżawy, a `Send-MailMessage` nigdy nie potrafiło zalogować się przez OAuth 2.0.

Po co to istnieje

Skrypt administracyjny prędzej czy później potrzebuje kogoś powiadomić — że backup się nie powiódł, że dysk się zapełnia, że audyt z poprzedniej strony wykrył coś wartego uwagi. `Send-MailMessage` przez lata było jedynym wbudowanym w PowerShella poleceniem robiącym to bez instalowania niczego dodatkowego, stąd jego wszechobecność w istniejących skryptach mimo formalnego statusu przestarzałego.

> **Pojęcie: STARTTLS.** Mechanizm, w którym połączenie SMTP zaczyna się jako zwykłe, nieszyfrowane połączenie na porcie 587, a dopiero po nawiązaniu klient i serwer negocjują przejście na szyfrowanie TLS w trakcie tej samej sesji, zanim padnie jakiekolwiek polecenie z danymi uwierzytelniającymi albo treścią wiadomości. To odróżnia port 587 od portu 465 (SMTPS), gdzie szyfrowanie TLS obejmuje połączenie od pierwszego bajtu, bez etapu negocjacji w locie. `-UseSsl` w `Send-MailMessage` uruchamia właśnie STARTTLS na porcie 587 — mimo nazwy parametru sugerującej czyste SSL, w praktyce to najczęściej STARTTLS, bo to ten wariant obsługuje zdecydowana większość serwerów SMTP w 2026 roku.



Skrypt laczy sie na port 587, negocjuje STARTTLS z serwerem SMTP i dopiero wtedy dostarcza wiadomosc (kliknięcie otwiera pełny rozmiar)

Jak to działa

Powód formalnego wycofania `Send-MailMessage` przez Microsoft nie jest kosmetyczny — to polecenie nie potrafi uwierzytelnić się przez OAuth 2.0, a to jest dziś jedyny mechanizm, który Microsoft 365 akceptuje domyślnie dla nowych skrzynek. Skrypty, które nadal chcą wysyłać pocztę przez Microsoft 365 z uwierzytelnieniem loginem i hasłem, wymagają jawnie włączonego wyjątku uwierzytelniania podstawowego dla konkretnej skrzynki (coraz rzadziej dostępnego w nowych dzierżawach) albo przekierowania przez lokalny przekaźnik SMTP, który sam obsłuży uwierzytelnienie OAuth po swojej stronie.

Czym zastępujemy Send-MailMessage. Trzy realistyczne kierunki, w zależności od tego, dokąd wysyłamy pocztę i jak bardzo skrypt ma być przyszłościowy:

Zamiennik	Kiedy sięgamy
MailKit (biblioteka .NET)	skrypt PowerShellowy wysyłający przez dowolny serwer SMTP z pełnym wsparciem nowoczesnego uwierzytelniania i szyfrowania, bez zależności od konkretnego dostawcy poczty
Microsoft Graph API	wysyłka przez skrzynkę Microsoft 365/Exchange Online z uwierzytelnieniem OAuth 2.0, zgodnie z kierunkiem, w jaki idzie sam Microsoft
`curl` z SMTP	najprostsze, jednorazowe wysłanie z poziomu skryptu bez dociągania żadnej dodatkowej biblioteki — `curl` obsługuje protokół SMTP natywnie od dawna

Wysłanie wiadomości przez curl z protokołem SMTP

```
curl.exe --url "smtps://AdresSerwera:465" `
    --ssl-reqd `
    --mail-from "AdresNadawcy@NazwaDomeny" `
    --mail-rcpt "AdresOdbiorcy@NazwaDomeny" `
    --user "NazwaKontaSMTP:HasloDoSkrzynki" `
    --upload-file "SciezkaDoPliku.eml"
```

`curl` wymaga gotowej treści wiadomości w formacie `.eml` (nagłówki `From`/`To`/`Subject` i treść, rozdzielone pustą linią) podanej przez `--upload-file` — nie buduje wiadomości za nas tak, jak robił to `Send-MailMessage` przez parametry `-Subject`/`-Body`. To rozwiązanie sprawdza się jako szybki test łączności z serwerem SMTP albo jako awaryjny sposób wysyłki tam, gdzie nie chcemy instalować żadnej dodatkowej biblioteki .NET.

Częste błędy

Kontynuowanie użycia `Send-MailMessage` w nowym skrypcie bez świadomości, że jest przestarzałe. Działa dziś, ale opiera się na bibliotece, której Microsoft już nie rozwija — nowy skrypt pisany od zera, zwłaszcza jeśli ma wysyłać przez Microsoft 365, oszczędza sobie migracji w przyszłości, jeśli od razu sięgnie po MailKit albo Graph API.

Brak `-UseSsl` przy porcie 587. Serwer SMTP odrzuca próbę uwierzytelnienia w otwartym, nieszyfrowanym połączeniu, zanim w ogóle sprawdzi poprawność hasła — komunikat błędu bywa mylący i sugeruje problem z poświadczeniami, podczas gdy przyczyną jest brak negocjacji STARTTLS.

Umieszczenie hasła do skrzynki SMTP wprost w treści skryptu. Dokładnie ten sam problem, co opisany na stronie Poświadczenia i bezpieczne hasła w skryptach — konto SMTP używane tylko do wysyłki automatycznych powiadomień jest równie atrakcyjnym celem jak każde inne, a wyciek pozwala rozsyłać pocztę w imieniu organizacji.

□□ Ćwiczenia

>

> Ćwiczenia poniżej wymagają dostępu do testowego konta SMTP — nie wysyłamy testowej poczty przez produkcyjną skrzynkę firmową bez wyraźnej potrzeby.

Ćwiczenie 1 — zbudowanie wiadomości krok po kroku

```
$Credential = Get-Credential -Message "Poświadczenia do testowego konta SMTP"

Send-MailMessage -From "test@zielona.example" -To "test@zielona.example" `
    -Subject "Test $(Get-Date -Format 'yyyy-MM-dd HH:mm')" `
    -Body "Wiadomość testowa wysłana ze skryptu." `
    -SmtpServer "AdresSerwera" -Port 587 -Credential $Credential -UseSsl
```

Wysyłamy do siebie samych wiadomość testową z dynamicznym tematem zawierającym bieżącą datę i godzinę.

Odpowiedź

Jeśli polecenie zwraca błąd `Unable to connect to the remote server`, problem leży w łączności sieciowej albo w nazwie serwera SMTP — warto sprawdzić `Test-NetConnection -ComputerName AdresSerwera -Port 587` ze strony Sieć z wiersza poleceń przed dalszym diagnozowaniem. Jeśli połączenie się udaje, ale uwierzytelnienie zawodzi, przyczyną bywa brak `-UseSsl` albo konto SMTP wymagające nowoczesnego uwierzytelniania, którego `Send-MailMessage` nie obsługuje.

Ćwiczenie 2 — porównanie portów

```
Test-NetConnection -ComputerName "AdresSerwera" -Port 587
Test-NetConnection -ComputerName "AdresSerwera" -Port 465
Test-NetConnection -ComputerName "AdresSerwera" -Port 25
```

Sprawdzamy, które z trzech klasycznych portów SMTP są w ogóle otwarte na danym serwerze pocztowym.

Odpowiedź

Port `587` z STARTTLS jest dziś standardowym portem do wysyłki uwierzytelnionej poleceniem klienckim. Port `465` (SMTPS) zakłada szyfrowanie TLS od pierwszego bajtu połączenia, bez etapu negocjacji. Port `25` jest tradycyjnie zarezerwowany dla komunikacji serwer-serwer i coraz częściej blokowany dla ruchu wychodzącego z sieci klienckich przez dostawców internetu właśnie po to, żeby utrudnić rozsyłanie spamu bezpośrednio z zainfekowanych stacji roboczych.

Pytania kontrolne

1. Dlaczego skrypt wysyłający pocztę przez konto Microsoft 365 przestał nagle działać, mimo że hasło się nie zmieniło?

Odpowiedź

Najbardziej prawdopodobną przyczyną jest wygaszenie uwierzytelniania podstawowego (Basic Authentication) po stronie Microsoft 365 dla tej dzierżawy albo tej konkretnej skrzynki — `Send-MailMessage` loguje się loginem i hasłem, nie potrafi przeprowadzić uwierzytelnienia OAuth 2.0, którego Microsoft dziś wymaga domyślnie. Rozwiązaniem jest migracja na Graph API z uwierzytelnieniem aplikacyjnym albo skonfigurowanie lokalnego przekaźnika SMTP obsługującego OAuth po swojej stronie.

2. Czym różni się port 465 od portu 587 w kontekście szyfrowania połączenia SMTP?

Odpowiedź

Port `465` (SMTPS) zakłada połączenie szyfrowane TLS od samego początku, zanim padnie jakiekolwiek polecenie protokołu SMTP. Port `587` zaczyna połączenie jawnym tekstem i dopiero w jego trakcie klient i serwer negocjują przejście na szyfrowanie przez polecenie `STARTTLS` — `-UseSsl` w `Send-MailMessage` na porcie 587 uruchamia właśnie ten drugi mechanizm, mimo że nazwa parametru sugeruje czyste SSL/TLS od początku.

Powiązane: Poświadczenia i bezpieczne hasła w skryptach · Sieć z wiersza poleceń · Skrypty PowerShell: struktura i dobre praktyki

Źródła: `Get-Help Send-MailMessage` · dokumentacja Microsoft Learn, „Send-MailMessage is not recommended" i „MailKit documentation"