# CMD, PowerShell, PowerShell Core: trzy powłoki, trzy różne światy

**Poziom: podstawy** **Dotyczy: Windows 10/11, Windows Server 2016+** **Czas: ~20 min**

*Na jednym Windowsie żyją obok siebie trzy powłoki wiersza poleceń — ta strona tłumaczy, czym się różnią, i dlaczego PowerShell nie jest „lepszym cmd", tylko zupełnie inną koncepcją przekazywania danych między poleceniami.*

**Skróty użyte na tej stronie (2)**

| Skrót | Co znaczy |
|---|---|
| **BOM** | Byte Order Mark: znacznik na początku pliku tekstowego UTF-8, który dopisuje Excel |
| **UAC** | User Account Control: okienko Windows pytające o podniesienie uprawnień |

# Fiszka

**TYLKO ODCZYT**

## Sprawdzenie wersji cmd.exe: ver

```
ver
```

Wypisujemy numer wersji jądra systemu, na którym stoi `cmd.exe` — samo `cmd.exe` nie ma własnego numeru wersji niezależnego od systemu operacyjnego.

Przykład:

```
Microsoft Windows [Version 10.0.19045.4291]
```

🔒 TYLKO ODCZYT

## Sprawdzenie wersji Windows PowerShell: $PSVersionTable

```
$PSVersionTable
```

Zmienna automatyczna `$PSVersionTable` istnieje w każdej sesji PowerShella i mówi, z którą edycją silnika mamy do czynienia — `Desktop` to Windows PowerShell 5.1, `Core` to PowerShell 7+.

Przykład:

```
Name                           Value
----                           -----
PSVersion                      5.1.19041.4291
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.19041.4291
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

🔒 TYLKO ODCZYT

## Sprawdzenie, czy zainstalowany jest PowerShell 7: pwsh -Version

```
pwsh -Version
```

`pwsh.exe` to plik wykonywalny PowerShell 7+ — instalujemy go osobno, obok wbudowanego Windows PowerShell, nigdy w jego miejsce.

Przykład:

```
PowerShell 7.4.2
```

Jeśli `pwsh` nie jest zainstalowany, powłoka zwraca `pwsh: The term 'pwsh' is not recognized...` — to najszybszy test obecności PowerShell 7 na maszynie.

🔒 TYLKO ODCZYT

## Uruchomienie PowerShell 7 z poziomu Windows PowerShell

```
pwsh
```

Wchodzimy do zagnieżdżonej sesji PowerShell 7 wewnątrz sesji Windows PowerShell 5.1 — `exit` wraca do sesji nadrzędnej, nie zamyka okna.

**□□ ZMIENIA SYSTEM**

## Uruchomienie konsoli z podwyższonymi uprawnieniami: Start-Process -Verb RunAs

```
Start-Process powershell -Verb RunAs
```

Otwieramy nowe okno PowerShell z podniesionymi uprawnieniami (UAC), z poziomu sesji, która sama podniesionych uprawnień nie ma — Windows pyta o potwierdzenie w oknie kontroli konta użytkownika.

□□ **UAC** = User Account Control: okienko Windows pytające o podniesienie uprawnień

**□□ TYLKO ODCZYT**

## Sprawdzenie, czy bieżąca sesja jest podniesiona

```
([Security.Principal.WindowsPrincipal][Security.Principal.WindowsIdentity]::GetCurrent()
).IsInRole([Security.Principal.WindowsBuiltinRole]::Administrator)
```

Sprawdzamy programowo, bez otwierania nowego okna, czy bieżąca sesja działa z uprawnieniami administratora — przydatne na początku skryptu, który wymaga podniesienia.

Przykład:

```
True
```

**□□ TYLKO ODCZYT**

## Zajrzenie do wnętrza obiektu: Get-Member

```
Get-Process -Name explorer | Get-Member
```

`Get-Member` (alias `gm`) pokazuje, z jakim typem obiektu naprawdę mamy do czynienia w potoku — jego właściwości (`Property`) i metody (`Method`) — zamiast zgadywać na podstawie tego, co się wyświetliło na ekranie.

Przykład:

```
   TypeName: System.Diagnostics.Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Kill                       Method         void Kill(), void Kill(bool entireProcessTree)
Refresh                    Method         void Refresh()
WaitForExit                Method         bool WaitForExit(int milliseconds), void
WaitForExit()
Id                         Property       int Id {get;}
ProcessName                Property       string ProcessName {get;}
StartTime                  Property       datetime StartTime {get;}
WorkingSet64               Property       long WorkingSet64 {get;}
```

**□□ TYLKO ODCZYT**

## Potok obiektowy na przykładzie

```
Get-Process | Where-Object CPU -gt 50 | Select-Object ProcessName, Id, CPU | Sort-Object
CPU -Descending
```

Każdy element potoku dostaje na wejściu i oddaje na wyjściu prawdziwy obiekt `System.Diagnostics.Process`, nie tekst — `Where-Object` filtruje po właściwości `CPU`, a nie po pozycji kolumny w wierszu tekstu, więc kolejność ani szerokość kolumn na ekranie nie ma żadnego znaczenia dla działania filtra.

Przykład:

```
ProcessName          Id        CPU
-----------          --        ---
chrome             8420    412.34
MsMpEng            2104    198.77
sqlservr           5560    143.21
```

□□ **CPU** = Central Processing Unit: procesor

**🔒 TYLKO ODCZYT**

### Wszystkie trzy powłoki obok siebie

| Powłoka | Plik wykonywalny | Domyślnie w systemie | Silnik |
|---|---|---|---|
| CMD | `cmd.exe` | tak, od DOS-u | interpreter wsadowy, brak silnika obiektowego |
| Windows PowerShell | `powershell.exe` | tak, od Windows 7 / Server 2008 R2 | .NET Framework |
| PowerShell 7+ | `pwsh.exe` | nie, instalacja osobna | .NET (Core), wieloplatformowy |

**Uważaj**

- `powershell.exe` i `pwsh.exe` to dwa osobne pliki wykonywalne, dwa osobne profile, czasem dwa różne zestawy zainstalowanych modułów — sprawdzenie czegoś w jednym nie gwarantuje tego samego w drugim.
- Skrypt `.ps1` napisany i przetestowany pod PowerShell 7 nie zawsze uruchomi się bez poprawek pod Windows PowerShell 5.1 — część poleceń (na przykład operator `??`, `ternary`, niektóre parametry `Join-String`) istnieje tylko w nowszym silniku.
- `ISE` (Windows PowerShell Integrated Scripting Environment) obsługuje wyłącznie Windows PowerShell 5.1 i nie widzi PowerShell 7 — od kilku lat Microsoft rekomenduje w jego miejsce Visual Studio Code z rozszerzeniem PowerShell.

# Z życia wzięte

**🔒 Z ŻYCIA WZIĘTE**

Tomek, drugi tydzień w pracy, loguje się zdalnie na wsus01 we wtorek po Patch Tuesday i wkleja do okna PowerShell gotowy fragment z instrukcji dla cmd.exe, `if exist plik.txt echo istnieje`. Konsola odpowiada błędem składni, bo `if` w PowerShellu wymaga nawiasów i osobnego polecenia `Test-Path`. Tomek zamyka okno, przekonany że coś jest zepsute na serwerze, zamiast zauważyć, że stoi w zupełnie innej powłoce niż ta, dla której instrukcja była pisana.

# Po co to istnieje

> **Pojęcie: powłoka.** Program, który czyta polecenia od użytkownika albo ze skryptu, uruchamia odpowiadające im programy albo funkcje wbudowane i pokazuje wynik — pośrednik między człowiekiem a jądrem systemu. `cmd.exe`, `powershell.exe` i `pwsh.exe` to trzy niezależne implementacje tej samej roli na Windowsie, z zupełnie różnymi silnikami wewnątrz.

`cmd.exe` jest bezpośrednim potomkiem `COMMAND.COM` z DOS-u — interpreterem poleceń wsadowych, który od dekad utrzymuje kompatybilność wsteczną kosztem możliwości. Rozumie zmienne środowiskowe, potok `|`, przekierowania `>`/`>>`, proste pętle `for` i etykiety `goto`, ale każda komenda w potoku przekazuje następnej wyłącznie tekst — nic więcej. Nadal jest obecny, bo tysiące starych plików `.bat` i instalatorów go zakłada, i bo pewne narzędzia systemowe (`sc.exe`, `netsh`, `robocopy`) wciąż wygodniej wywołać wprost, niż przez otoczkę PowerShellową.

Windows PowerShell 5.1 pojawił się jako odpowiedź na ograniczenia `cmd.exe` — oparty o .NET Framework, wbudowany w każdy Windows od Windows 7 / Server 2008 R2 wzwyż, z własnym silnikiem obiektowym opisanym niżej. Jest to ostatnia wersja klasycznego, zamkniętego w Windowsie PowerShella — Microsoft rozwija ją już tylko poprawkami bezpieczeństwa, nie nowymi funkcjami.

PowerShell 7+ (nazywany też PowerShell Core, plik `pwsh.exe`) to następca zbudowany na .NET zamiast .NET Framework — wieloplatformowy (Windows, Linux, macOS), instalowany osobno, rozwijany aktywnie, z nowszą składnią i szybszym silnikiem. Nie zastępuje Windows PowerShell 5.1 w systemie — instaluje się obok, jako zupełnie osobny produkt z własnym profilem i własnym zestawem modułów.

# Analogia

`cmd.exe` działa jak wymiana karteczek z odręcznym tekstem między dwoma urzędnikami — jeden zapisuje wynik swojej pracy na kartce, podaje drugiemu, a ten musi najpierw odczytać i zinterpretować, co tam jest napisane, zanim zacznie własną robotę. Jeśli format karteczki się zmieni choć trochę — inna liczba spacji, inna kolejność kolumn — drugi urzędnik się gubi.

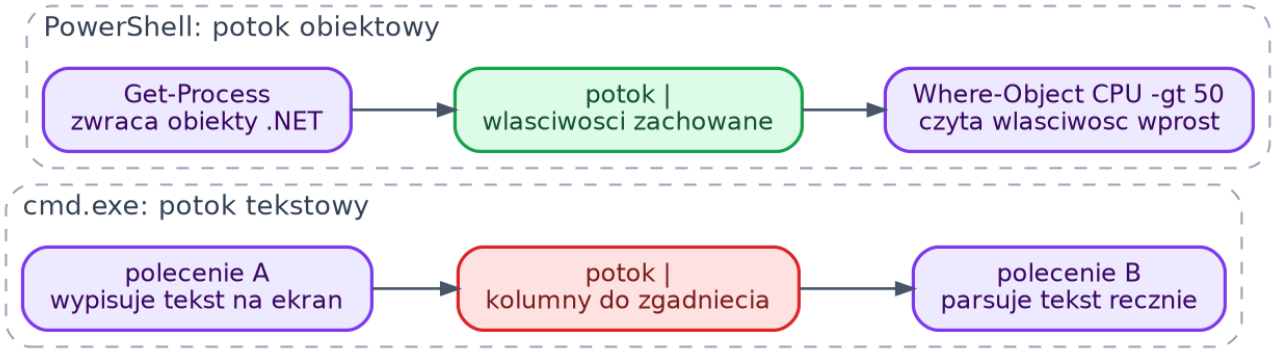
PowerShell działa jak przekazywanie sobie gotowych, podpisanych segregatorów z danymi. Segregator ma ustaloną strukturę: pole „nazwa", pole „rozmiar", pole „data utworzenia" — drugi urzędnik nie musi niczego parsować, sięga bezpośrednio po pole, które go interesuje, niezależnie od tego, w jakiej kolejności segregator wyświetla te pola na okładce. Ograniczenie tej analogii: segregator trzeba było najpierw kimś wypełnić zgodnie ze wspólnym standardem — obiekty w PowerShellu istnieją, bo producent polecenia (`cmdlet`) świadomie zwraca ustandaryzowany typ

.NET, a nie dowolny tekst. Polecenie napisane niedbale, które zwraca gołe stringi zamiast obiektów, psuje tę wygodę tak samo jak w cmd.exe.

# Jak to działa

**Rozwiń: jak to działa pod spodem**

> **Pojęcie: potok obiektowy.** W bashu i w cmd.exe potok `|` przekazuje strumień bajtów — jedno polecenie musi wypisać tekst w formacie, który drugie polecenie potrafi sparsować (kolumny rozdzielone spacjami, wartości w konkretnej kolejności), a każda zmiana formatu w nowszej wersji narzędzia potrafi cichutko połamać skrypt korzystający z tego tekstu. W PowerShellu `|` przekazuje żywe obiekty .NET, zachowujące pełną strukturę — typ, wszystkie właściwości, metody — aż do chwili, gdy coś świadomie zamieni je na tekst (najczęściej `Format-Table`, `Out-File` albo po prostu koniec potoku i wypisanie na ekran). To jest fundamentalna różnica względem basha: `ls -la | awk '{print $5}'` musi zgadywać, w której kolumnie tekstu leży rozmiar pliku, podczas gdy `Get-ChildItem | Select-Object Length` sięga wprost po właściwość `Length` obiektu `FileInfo`, niezależnie od tego, jak szeroko akurat wyrenderowała się kolumna na ekranie.




*Potok tekstowy w cmd.exe zmusza do zgadywania kolumn, potok obiektowy w PowerShellu przenosi wlasciwosci wprost (kliknięcie otwiera pełny rozmiar)*

> **Pojęcie: moduł.** Paczka poleceń PowerShella (`cmdlet`ów), funkcji i zmiennych, którą ładujemy do sesji poleceniem `Import-Module`, żeby uzyskać dostęp do konkretnego

obszaru funkcjonalności — moduł `ActiveDirectory` daje polecenia typu `Get-ADUser`, moduł `NetSecurity` daje `Get-NetFirewallRule`. Windows PowerShell 5.1 i PowerShell 7 mają w dużej mierze wspólne moduły wbudowane w system, ale moduły instalowane ręcznie z galerii (`Install-Module`) trafiają do osobnych, niepowiązanych ze sobą lokalizacji dla każdego z silników.

**Pojęcie: profil.** Skrypt `.ps1` wykonywany automatycznie przy starcie sesji PowerShella — miejsce na własne aliasy, funkcje pomocnicze i `Import-Module` poleceń, których i tak wpisywalibyśmy ręcznie po każdym uruchomieniu. Ścieżka profilu jest dostępna pod zmienną automatyczną `$PROFILE` i jest inna dla każdej kombinacji silnika, hosta (konsola kontra ISE kontra VS Code) i zakresu (bieżący user kontra wszyscy userzy):

| Silnik | Przykładowa ścieżka profilu bieżącego usera |
|---|---|
| Windows PowerShell 5.1 | `C:\Users\NazwaUzytkownika\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1` |
| PowerShell 7+ | `C:\Users\NazwaUzytkownika\Documents\PowerShell\Microsoft.PowerShell_profile.ps1` |

Różnica w nazwie folderu (`WindowsPowerShell` kontra `PowerShell`) jest zamierzona i pozwala obu silnikom trzymać osobne profile bez wzajemnego nadpisywania.

**Kodowanie.** Windows PowerShell 5.1 domyślnie zapisuje pliki poleceniem `Out-File` w UTF-16 z BOM, a odczytuje `cmd.exe` w bieżącej stronie kodowej konsoli (najczęściej `850` albo `852` na polskich systemach) — stąd polskie znaki poprawnie wpisane w PowerShellu potrafią wyjść jako krzaki po otwarciu tego samego pliku w `cmd.exe` albo w edytorze bez wsparcia BOM. PowerShell 7 domyślnie używa UTF-8 bez BOM, co bliżej odpowiada temu, czego oczekuje reszta współczesnych narzędzi — ale przy migracji skryptu ze starszego silnika na nowszy kodowanie plików wyjściowych jest jednym z pierwszych miejsc, które warto sprawdzić ręcznie.

# Częste błędy

**Wklejanie składni cmd.exe do skryptu PowerShell.** `if exist plik.txt echo istnieje` w oknie PowerShella zwraca błąd składni — `if` w PowerShellu wymaga nawiasów i bloku `{}`, a test istnienia pliku to osobne polecenie `Test-Path`.

**Poleganie na aliasach w skrypcie, który ktoś inny uruchomi pod inną wersją.** Alias `ls` w Windows PowerShell wskazuje na `Get-ChildItem`, ale w PowerShell 7 na Linuksie `ls` bywa też prawdziwym binarnym poleceniem systemowym z zupełnie innym zestawem flag — pełne omówienie na stronie Podstawy składni PowerShella.

**Uruchomienie polecenia wymagającego podniesienia bez `Start-Process -Verb RunAs`.** Skrypt kończy się komunikatem `Access to the path is denied` albo `The requested operation requires elevation`, mimo że logujemy się na konto z uprawnieniami administratora — samo posiadanie uprawnień nie wystarcza, sesja musi być świadomie podniesiona przez UAC.

**Mylenie PowerShell ISE z PowerShell 7.** ISE ma niebieskie tło i wygląda znajomo starszym administratorom, ale to środowisko wyłącznie dla Windows PowerShell 5.1 — polecenia dostępne tylko w PowerShell 7 (na przykład operator `?:`) w ISE zwrócą błąd składni, nawet jeśli `pwsh` jest zainstalowany na tej samej maszynie.

# □□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell, bez uprawnień administratora, chyba że napisano inaczej.

## Ćwiczenie 1 — porównanie tekstu i obiektu

```
$procesy = Get-Process
$procesy | Get-Member -MemberType Property | Select-Object -First 5
$procesy | Format-Table Name, Id, CPU -AutoSize | Out-String | Get-Member
```

Dlaczego drugie polecenie w trzeciej linii zwraca zupełnie inny, dużo krótszy zestaw właściwości niż pierwsze `Get-Member`?

**Odpowiedź**

`Format-Table` przekształca żywe obiekty `Process` w gotowy do wyświetlenia tekst tabeli — `Out-String` sklaja go w pojedynczy ciąg znaków. Od tego momentu w potoku nie ma już żadnych właściwości procesu, jest wyłącznie napis, więc `Get-Member` widzi typ `System.String` z jego własnymi metodami, a nie strukturę procesu. To pokazuje wprost, dlaczego `Format-*` musi stać na samym końcu potoku — po nim obiektowość znika bezpowrotnie.

## Ćwiczenie 2 — sprawdzenie własnej powłoki

```
$PSVersionTable.PSEdition
$PSVersionTable.PSVersion
$PROFILE
Test-Path $PROFILE
```

Sprawdzamy, w jakim silniku właśnie pracujemy i czy mamy już zapisany plik profilu.

**Odpowiedź**

`PSEdition` równe `Desktop` oznacza sesję Windows PowerShell 5.1, `Core` oznacza PowerShell 7+. `$PROFILE` zwraca ścieżkę specyficzną dla tej konkretnej kombinacji silnika i hosta — `Test-Path $PROFILE` zwracające `False` nie jest błędem, oznacza tylko, że plik profilu nigdy nie został utworzony, a domyślne zachowanie powłoki jest wtedy stosowane bez żadnych własnych dodatków.

# Pytania kontrolne

**1. Dlaczego `Get-Process | Where-Object CPU -gt 50` działa poprawnie niezależnie od tego, w jakiej kolejności `Get-Process` wypisze kolumny na ekranie?**

**Odpowiedź**

`Where-Object` filtruje po nazwanej właściwości `CPU` obiektu `Process`, nie po pozycji tekstu w kolumnie — kolejność i szerokość kolumn to sprawa wyłącznie sposobu wyświetlenia na ekranie, ustalana dopiero przez ewentualny `Format-Table` na końcu potoku, i nie ma żadnego wpływu na dane, którymi operują wcześniejsze polecenia w łańcuchu.

**2. Dlaczego zainstalowanie PowerShell 7 na serwerze nie usuwa ryzyka, że stary skrypt `.ps1` z crona przestanie działać poprawnie?**

**Odpowiedź**

PowerShell 7 instaluje się obok Windows PowerShell 5.1, nie w jego miejsce — który silnik faktycznie uruchomi zaplanowane zadanie, zależy od tego, jaki plik wykonywalny ( `powershell.exe` czy `pwsh.exe`) wskazano w akcji zadania. Jeśli skrypt korzysta ze składni albo modułu dostępnego tylko w jednym z silników, zmiana silnika uruchamiającego bez przetestowania skryptu pod nim jest realnym źródłem awarii.

**Powiązane:** Podstawy składni PowerShella · Polityka wykonywania i podpisywanie skryptów · Skrypty PowerShell: struktura i dobre praktyki

**Źródła:** `Get-Help about_Pipelines` · `Get-Help about_Profiles` · dokumentacja Microsoft Learn, „PowerShell editions"

---

Revision #38
Created 2026-09-25 06:44:27 UTC by Tomasz Konieczny
Updated 2026-10-18 08:53:11 UTC by Tomasz Konieczny