Instalacja i usuwanie oprogramowania

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

□ Czas: ~25 min

Cztery różne mechanizmy instalacji współistnieją na Windowsie naraz — pakiety MSI, opcjonalne komponenty systemu, role serwera i menedżer pakietów `winget` — i każdy ma inne polecenie do sprawdzenia, czy się powiodła.

□□ Skróty użyte na tej stronie (8)

Skrót	Co znaczy
CMD	skrót od polecenia command, dawna nazwa wiersza poleceń Windows sprzed PowerShella
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
GUID	Globally Unique Identifier: unikalny w skali globalnej identyfikator obiektu, tu konkretnego protektora klucza BitLockera
MSI	Microsoft Installer: format paczki instalacyjnej Windows
PDF	Portable Document Format: format dokumentu, który wygląda tak samo na każdym komputerze, np. zakres akredytacji jednostki na stronie PCA
RMM	Remote Monitoring and Management: platforma do zdalnego zarządzania i monitorowania stacji roboczych
RSAT	Remote Server Administration Tools: dodatkowe narzędzia administracyjne Windows do zarządzania AD, DNS i GPO ze stacji roboczej
WWW	World Wide Web: strony i aplikacje otwierane w przeglądarce po HTTP albo HTTPS

Fiszka

□□ ZMIENIA SYSTEM

Instalacja pakietu MSI w trybie cichym: msiexec /qn

```
msiexec /i "SciezkaDoPliku.msi" /qn /norestart /l*v "SciezkaDoLogu.log"
```

Uruchamiamy z poziomu CMD bądź PowerShell z uprawnieniami administratora. `/qn` (quiet, no UI) instaluje bez jakiegokolwiek okna dialogowego — żadnego kreatora, żadnego paska postępu, nic do kliknięcia, co jest warunkiem koniecznym do uruchomienia instalacji z poziomu skryptu bez nadzoru. `/norestart` wstrzymuje automatyczny restart systemu, nawet jeśli instalator normalnie by go zażądał — restart wykonujemy świadomie, osobnym poleceniem, w oknie serwisowym, nie jako efekt uboczny instalacji. `/l*v` włącza szczegółowe (`verbose`) logowanie całego przebiegu instalacji do wskazanego pliku.

MSI = Microsoft Installer: format paczki instalacyjnej Windows · **CMD** = skrót od polecenia command, dawna nazwa wiersza poleceń Windows sprzed PowerShella · **msiexec** = Microsoft Installer execute: instalacja paczek MSI

ZMIENIA SYSTEM

Uruchomienie instalatora MSI z poziomu PowerShella z oczekiwaniem na zakończenie

```
$proces = Start-Process msiexec.exe -ArgumentList '/i', '"SciezkaDoPliku.msi"', '/qn', '/norestart', '/l*v', '"SciezkaDoLogu.log"' -Wait -PassThru
$proces.ExitCode
```

> **Pojęcie: Start-Process -Wait -PassThru.** Bez `-Wait` `Start-Process` uruchamia zewnętrzny program i natychmiast oddaje sterowanie z powrotem do skryptu, nie czekając na jego zakończenie — kolejna linia skryptu (na przykład sprawdzenie, czy plik zainstalowanej aplikacji już istnieje) wykonałaby się, zanim instalator w ogóle skończy pracę. `-Wait` wstrzymuje wykonanie skryptu do chwili zamknięcia się procesu, a `-PassThru` sprawia, że `Start-Process` zwraca obiekt procesu (z właściwością `ExitCode`) zamiast nic nie zwracać — bez `-PassThru` `$proces` pozostałoby puste, mimo że `-Wait` zadziałało poprawnie.

MSI = Microsoft Installer: format paczki instalacyjnej Windows

TYLKO ODCZYT

Sprawdzenie kodu wyjścia instalacji

```
if ($proces.ExitCode -eq 0) {
    Write-Host "Instalacja zakończona powodzeniem" -ForegroundColor Green
```

```
} elseif ($proces.ExitCode -eq 3010) {
    Write-Host "Instalacja zakończona powodzeniem, wymagany restart" -ForegroundColor
Yellow
} else {
    Write-Host "Instalacja zakończona błędem, kod: $($proces.ExitCode)" -ForegroundColor
Red
}
```

> **Pojęcie: kod wyjścia MSI.** `0` oznacza pełny sukces, `3010` oznacza sukces wymagający restartu (odłożonego dzięki `/norestart` z fiszki wyżej), każdy inny kod jest błędem konkretnego typu udokumentowanym przez Microsoft w liście kodów błędów Windows Installer — samo `/l*v` z poprzedniej fiszki jest tym, co pozwala doczytać w pliku logu, co dokładnie poszło nie tak przy kodzie innym niż `0` czy `3010`.

Przykład:

```
Instalacja zakończona powodzeniem, wymagany restart
```

MSI = Microsoft Installer: format paczki instalacyjnej Windows

TYLKO ODCZYT

Sprawdzenie dostępnych komponentów opcjonalnych systemu: Get-WindowsCapability

```
Get-WindowsCapability -Online | Where-Object Name -like "*Telnet*"
```

> **Pojęcie: komponent opcjonalny (Windows Capability).** Funkcje systemu Windows dostarczane jako osobne, doinstalowywane na żądanie pakiety — klient Telnet, klient/serwer OpenSSH, narzędzia RSAT to typowe przykłady. `-Online` oznacza sprawdzenie stanu na uruchomionym, bieżącym systemie, w odróżnieniu od operowania na obrazie offline.

Przykład:

```
Name  : Telnet.Client~~~~0.0.1.0
State : NotPresent
```

□□ **RSAT** = Remote Server Administration Tools: dodatkowe narzędzia administracyjne Windows do zarządzania AD, DNS i GPO ze stacji roboczej

□□ ZMIENIA SYSTEM

Instalacja komponentu opcjonalnego: Add-WindowsCapability

```
Add-WindowsCapability -Online -Name "Telnet.Client~~~~0.0.1.0"
```

Instalujemy dokładnie wskazany komponent po jego pełnej nazwie zwróconej przez `Get-WindowsCapability` — nazwa zawiera wersję i architekturę, więc kopiujemy ją zamiast wpisywać z pamięci.

Przykład:

```
Path          :
Online        : True
RestartNeeded : False
```

□□ TYLKO ODCZYT

Instalacja Telnet przez starsze narzędzie: pkgmgr

```
pkgmgr /iu:"TelnetClient"
```

Komendę uruchamiamy jako Administrator w CMD bądź PowerShell. `pkgmgr` jest starszym, poprzedzającym `Add-WindowsCapability` narzędziem instalacji komponentów opcjonalnych — na nowszych wersjach Windows 10/11 bywa już niedostępne albo oznaczone jako przestarzałe, `Add-WindowsCapability` jest dziś właściwym wyborem na nowo pisanym skrypcie. Zostawiamy to polecenie jako wiedzę pomocniczą do rozpoznania go w starszych, istniejących skryptach.

□□ **CMD** = skrót od polecenia command, dawna nazwa wiersza poleceń Windows sprzed PowerShella

□□ ZMIENIA SYSTEM

Instalacja klienta i serwera OpenSSH

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
```

Ten sam mechanizm co przy Telnecie — OpenSSH od Windows Server 2019 / Windows 10 1809 jest dostarczany jako komponent opcjonalny wbudowany w system, bez potrzeby pobierania zewnętrznego instalatora.

□□ ZMIENIA SYSTEM

Instalacja narzędzi RSAT na kliencie Windows

```
Get-WindowsCapability -Online | Where-Object Name -like "Rsat.ActiveDirectory*" | Add-WindowsCapability -Online
```

Przepuszczamy wynik wyszukania przez potok wprost do `Add-WindowsCapability` — wygodne, gdy nie chcemy przepisywać pełnej, długiej nazwy komponentu ręcznie. RSAT (Remote Server Administration Tools) na stacji roboczej daje dostęp do modułów takich jak `ActiveDirectory`, wykorzystywanych na stronie Skrypty audytowe Active Directory.

□□ **RSAT** = Remote Server Administration Tools: dodatkowe narzędzia administracyjne Windows do zarządzania AD, DNS i GPO ze stacji roboczej

□□ ZMIENIA SYSTEM

Włączenie funkcji opcjonalnej systemu klienckiego: Enable-WindowsOptionalFeature

```
Enable-WindowsOptionalFeature -Online -FeatureName "Microsoft-Hyper-V-All" -All -NoRestart
```

> **Pojęcie: funkcja opcjonalna kontra komponent (Capability).** Funkcje opcjonalne (`Windows Optional Feature`) to inny, starszy mechanizm niż komponenty opcjonalne (`Windows Capability`) — obejmuje większe, bardziej złożone podsystemy, takie jak Hyper-V, .NET Framework 3.5 czy kontener Windows Sandbox, zarządzane osobnym poleceniem i osobną listą niż `Get-WindowsCapability`. Oba mechanizmy współistnieją na tym samym systemie i trzeba wiedzieć, do którego z nich należy dana funkcja, żeby użyć właściwego polecenia.

□□ TYLKO ODCZYT

Sprawdzenie stanu funkcji opcjonalnej

```
Get-WindowsOptionalFeature -Online -FeatureName "Microsoft-Hyper-V-All"
```

Przykład:

```
FeatureName : Microsoft-Hyper-V-All
State       : Enabled
```

□□ ZMIENIA SYSTEM

Instalacja roli serwera: Install-WindowsFeature

```
Install-WindowsFeature -Name "DNS", "RSAT-DNS-Server" -IncludeManagementTools
```

> **Pojęcie: rola i funkcja serwera.** Na Windows Server rozróżniamy role (większe zestawy funkcjonalności definiujące przeznaczenie serwera — kontroler domeny, serwer DNS, serwer plików) i funkcje (mniejsze, uzupełniające komponenty) — `Install-WindowsFeature` obsługuje oba rodzaje jednym poleceniem. `-IncludeManagementTools` dokłada konsolę zarządzania i moduły PowerShella potrzebne do administrowania rolą, nie tylko samą jej działającą funkcjonalność.

Przykład:

```
Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {DNS, Remote Server Administration Tools...}
```

□□ **DNS** = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP

□□ TYLKO ODCZYT

Sprawdzenie zainstalowanych ról i funkcji na serwerze

```
Get-WindowsFeature | Where-Object Installed -eq $true | Select-Object Name, DisplayName
```

Przykład:

```
Name                       DisplayName
----                       -----------
DNS                        DNS Server
```

```
FS-FileServer                File and Storage Services
Web-Server                   Web Server (IIS)
```

◫◫ ZMIENIA SYSTEM

Instalacja pakietu przez winget

```
winget install --id NazwaPakietu --silent --accept-package-agreements --accept-source-agreements
```

> **Pojęcie: winget.** Menedżer pakietów wbudowany w Windows 10/11 (i doinstalowywany na Windows Server), pobierający i instalujący oprogramowanie z centralnego repozytorium Microsoftu i repozytoriów firm trzecich, jednym poleceniem, bez ręcznego pobierania instalatora ze strony producenta. `--silent` odpowiada w duchu `/qn` z `msiexec`, `--accept-package-agreements` / `--accept-source-agreements` pomijają interaktywne pytania o akceptację licencji, co jest wymogiem do uruchomienia w skrypcie bez nadzoru.

◫◫ **winget** = Windows Package Manager: menedżer pakietów do instalacji programów w Windows

◫◫ TYLKO ODCZYT

Sprawdzenie zainstalowanych pakietów: Get-Package

```
Get-Package -Name "*NazwaPakietu*" | Select-Object Name, Version, ProviderName
```

> **Pojęcie: Get-Package.** Ogólne polecenie modułu `PackageManagement`, zwracające zainstalowane oprogramowanie niezależnie od tego, jakim mechanizmem zostało zainstalowane (MSI, `winget`, `Programs and Features`) — wygodne jako jeden punkt sprawdzenia zamiast pamiętania, którym narzędziem dany pakiet trafił na maszynę.

Przykład:

```
Name              Version      ProviderName
----              -------      ------------
seven-zip.7zip    23.01        msi
```

◫◫ **MSI** = Microsoft Installer: format paczki instalacyjnej Windows

◫◫ ZMIENIA SYSTEM

Odinstalowanie aplikacji po nazwie

```
Get-Package -Name "NazwaPakietu" | Uninstall-Package
```

Odinstalowujemy przez ten sam moduł `PackageManagement`, którym sprawdzaliśmy obecność pakietu — działa dla oprogramowania zainstalowanego mechanizmami, które ten moduł rozpoznaje (głównie MSI i pakiety `winget` zarejestrowane w standardowym rejestrze odinstalowywania).

◫◫ **MSI** = Microsoft Installer: format paczki instalacyjnej Windows

◫◫ ZMIENIA SYSTEM

Odinstalowanie po kodzie produktu MSI

```
msiexec /x {12345678-90AB-CDEF-1234-567890ABCDEF} /qn /norestart /l*v
"SciezkaDoLogu.log"
```

> **Pojęcie: kod produktu (ProductCode).** Unikalny identyfikator GUID przypisany każdemu pakietowi MSI przy jego budowaniu — pozwala odinstalować dokładnie tę wersję pakietu, nawet jeśli sam plik `.msi` użyty do instalacji już nie istnieje na dysku. Kod produktu odczytujemy z rejestru (`HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`) albo z właściwości `IdentifyingNumber` w `Get-CimInstance Win32_Product`, jeśli pakiet nie jest już nigdzie indziej widoczny.

◫◫ **MSI** = Microsoft Installer: format paczki instalacyjnej Windows · **GUID** = Globally Unique Identifier: unikalny w skali globalnej identyfikator obiektu, tu konkretnego protektora klucza BitLockera · **HKLM** = HKEY_LOCAL_MACHINE: gałąź rejestru Windows z ustawieniami całego komputera, a nie pojedynczego użytkownika · **msiexec** = Microsoft Installer execute: instalacja paczek MSI

◫◫ TYLKO ODCZYT

Odinstalowanie pakietu winget

```
winget uninstall --id NazwaPakietu --silent
```

Symetryczne do instalacji — `winget` pamięta, co sam zainstalował, i potrafi to samo cofnąć bez konieczności szukania kodu produktu ręcznie.

winget = Windows Package Manager: menedżer pakietów do instalacji programów w Windows

ZMIENIA SYSTEM

Instalacja MSI po cichu z logiem: msiexec /i ... /qn /norestart /l*v

```
msiexec /i "SciezkaDoPliku.msi" /qn /norestart /l*v "SciezkaDoLogu.log"
```

To dokładnie ta sama komenda co w pierwszej fiszce tej strony, warta osobnej karty jako gotowy wzorzec do wklejenia w zadaniu RMM: bez `REINSTALL`, bez `REINSTALLMODE`, z logiem `/l*v` włączonym od razu, nie dopiero po tym, jak coś przestanie działać. Sekcja niżej pokazuje, dlaczego dopisanie `REINSTALL=ALL` do tej samej komendy potrafi ją po cichu zepsuć.

Uważaj

- `Get-CimInstance Win32_Product` (starsze polecenie do listowania zainstalowanych pakietów MSI) wywołuje przy każdym zapytaniu ciche przeliczenie (`reconfiguration`) wszystkich zainstalowanych pakietów MSI w tle — potrafi zauważalnie spowolnić system i generować dodatkowe wpisy w dzienniku zdarzeń, `Get-Package` tego problemu nie ma.
- `pkgmgr` jest przestarzałym narzędziem — na nowo pisanych skryptach sięgamy po `Add-WindowsCapability`, zostawiając `pkgmgr` wyłącznie do zrozumienia starszego, istniejącego kodu.
- Kod wyjścia `3010` z instalacji MSI oznacza sukces wymagający restartu — traktowanie go jako błędu w automatyzacji jest częstą pomyłką, która przerywa dalszy przebieg wdrożenia, mimo że instalacja faktycznie się powiodła.

MSI = Microsoft Installer: format paczki instalacyjnej Windows · **RMM** = Remote Monitoring and Management: platforma do zdalnego zarządzania i monitorowania stacji roboczych · **msiexec** = Microsoft Installer execute: instalacja paczek MSI

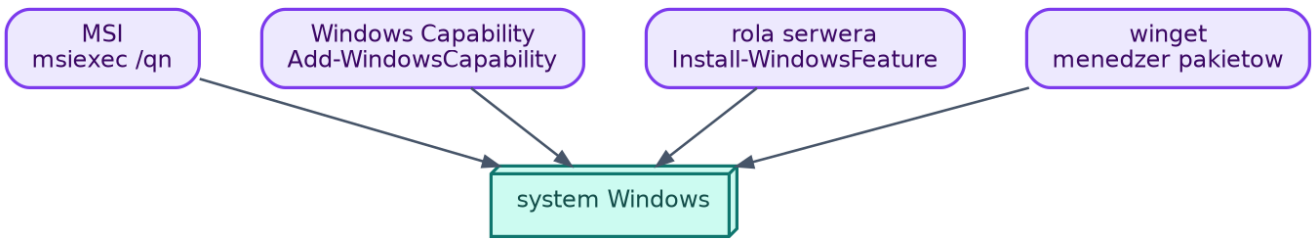
Z życia wzięte

Z ŻYCIA WZIĘTE

> Zlecamy przez RMM cichą aktualizację czytnika PDF na wszystkich stacjach w Pamiątkowie w środę wieczorem, z dopisanym na wszelki wypadek `REINSTALL=ALL`. Rano konsola RMM pokazuje same zielone haczyki i kod wyjścia `0` na każdej maszynie. Po południu okazuje się, że żadna stacja nie ma nowszej wersji, bo `REINSTALL=ALL` wyłączyło mechanizm `FindRelatedProducts`, który miał rozpoznać i usunąć starszy pakiet.

Po co to istnieje

Windows utrzymuje cztery częściowo niezależne mechanizmy dostarczania oprogramowania, bo powstawały w różnych erach systemu i obsługują różne kategorie potrzeb. MSI (Windows Installer) istnieje od Windows 2000 i pozostaje standardem dla aplikacji komercyjnych i wewnętrznych paczek firmowych — ma ustandaryzowany format, transakcyjną instalację (nieudana instalacja wycofuje wszystkie zmiany) i wbudowane wsparcie dla cichej instalacji, na czym opiera się cała pierwsza część tej strony. Komponenty i funkcje opcjonalne (`Windows Capability`, `Windows Optional Feature`) to mechanizm samego systemu operacyjnego do włączania fragmentów, które są już fizycznie na dysku (albo pobierane z Windows Update na żądanie), ale domyślnie wyłączone, żeby nie zajmować miejsca i powierzchni ataku niepotrzebnie. Role i funkcje serwera to rozszerzenie tego samego pomysłu, dostosowane do specyfiki Windows Server, gdzie „przeznaczenie" maszyny (kontroler domeny, serwer plików, serwer WWW) jest pierwszoplanową koncepcją zarządzania. `winget` jest najmłodszym z czterech — odpowiedzią na brak scentralizowanego menedżera pakietów w stylu tych znanych z Linuksa, pozwalającym zainstalować dowolną, zewnętrzną aplikację jednym poleceniem bez ręcznego szukania instalatora.



Cztery niezalezne mechanizmy dostarczania oprogramowania na Windowsie, kazdy z wlasnym poleceniem sprawdzajacym (kliknięcie otwiera pełny rozmiar)

Jak to działa

Różnica między `Get-WindowsCapability`/`Add-WindowsCapability` a `Get-WindowsOptionalFeature`/`Enable-WindowsOptionalFeature` bywa mylona, bo z zewnątrz obie robią „to samo" — włączają coś, co

leży uśpione w systemie. Rozróżnienie sprowadza się do tego, jak dana funkcjonalność jest zapakowana wewnętrznie: komponenty (`Capability`) są mniejszymi, częściej aktualizowanymi pakietami, pobieranymi na żądanie z Windows Update, jeśli nie są jeszcze obecne lokalnie — stąd Telnet czy OpenSSH, dodane relatywnie niedawno, poszły tą drogą. Funkcje opcjonalne (`Optional Feature`) są starszym mechanizmem, obejmującym zwykle większe, bardziej złożone podsystemy obecne w obrazie systemu od instalacji — stąd Hyper-V czy stary .NET Framework 3.5 wciąż korzystają z tego wzorca. Nie ma jednej reguły pozwalającej odgadnąć z góry, do którego mechanizmu należy dana funkcja — sprawdzamy oba polecenia, jeśli szukana nazwa nie pojawia się w pierwszym z nich.

Częste błędy

Traktowanie kodu wyjścia 3010 jako błędu instalacji. Skrypt wdrożeniowy, który sprawdza wyłącznie `if ($exitCode -ne 0)`, przerywa dalsze kroki mimo w pełni udanej instalacji — właściwe sprawdzenie uwzględnia explicite `0` i `3010` jako sukces, tak jak w fiszce wyżej.

Instalacja MSI bez `/norestart` w skrypcie wdrażającym wiele pakietów po kolei. Instalator z uprawnieniem do restartu systemu potrafi uruchomić restart natychmiast po swoim zakończeniu, przerywając w połowie instalację kolejnych pakietów w tej samej sekwencji skryptu.

Szukanie komponentu wyłącznie przez `Get-WindowsOptionalFeature`, gdy w rzeczywistości jest to `Windows Capability`. Polecenie zwraca pusty wynik albo błąd `The feature name is not recognized`, sugerując brak danej funkcji w systemie, podczas gdy w rzeczywistości trzeba było sprawdzić przez `Get-WindowsCapability` — oba mechanizmy trzeba sprawdzić osobno, dopóki nie wiemy z góry, do którego dana funkcja należy.

Wdrażanie MSI przez RMM

Zlecenie z RMM na „cichą aktualizację" pakietu potrafi zakończyć się sukcesem w konsoli RMM, a mimo to nie zmienić na komputerze zupełnie nic. Winny bywa jeden dopisany parametr, o którym instalator nie mówi żadnym błędem.

> Polecenie z dopisanym `REINSTALL=ALL` (często razem z `REINSTALLMODE=vomus` albo podobnym) wygląda jak bezpieczny sposób na wymuszenie ponownej instalacji, ale w rzeczywistości wyłącza logikę `FindRelatedProducts` i `RemoveExistingProducts`, którą pakiety budowane w `WiX` (`Windows Installer XML`) opierają na tabeli `Upgrade`. Bez tej logiki instalator nie rozpoznaje starszej wersji produktu jako czegoś do podmiany, `msiexec` kończy działanie kodem `0`, a na dysku zostaje dokładnie ta wersja, która była przed zleceniem.

Poprawne polecenie do „aktualizacji przez reinstalację" nie różni się niczym od zwykłej cichej instalacji nowszego pakietu:

```
msiexec /i "SciezkaDoNowszegoPliku.msi" /qn /norestart /l*v "SciezkaDoLogu.log"
```

Sam nowszy plik `.msi`, bez `REINSTALL` i bez `REINSTALLMODE`, pozwala instalatorowi samodzielnie rozpoznać starszą wersję przez `FindRelatedProducts` i usunąć ją przez `RemoveExistingProducts`, zanim zainstaluje nową. `REINSTALL=ALL` ma sens wyłącznie przy naprawie już zainstalowanej, tej samej wersji pakietu (na przykład gdy brakuje pliku po awarii dysku), nie przy podnoszeniu wersji.

Drugi, niezwiązany z `REINSTALL` kłopot z tej samej rodziny zleceń: niektóre RMM dławią się znakiem `|` w jednolinijkowym poleceniu cmd, obcinając wszystko po nim albo zgłaszając błąd składni, mimo że ta sama komenda uruchomiona ręcznie w oknie CMD działa bez zarzutu.

```
tasklist > "%TEMP%\lista-procesow.txt"
findstr /i "NazwaProcesu" "%TEMP%\lista-procesow.txt"
```

Zamiast łączyć dwie komendy potokiem `|` w jednym poleceniu zlecanym przez RMM, kierujemy wynik pierwszej komendy do pliku tymczasowego, a potem przeszukujemy ten plik osobnym poleceniem `findstr`: dwa proste kroki zamiast jednego, który RMM potrafi po drodze uszkodzić.

□□ Ćwiczenia

> Wszystko poniżej wymaga uprawnień administratora. Instalację i deinstalację realnych pakietów testujemy na maszynie testowej, nie na stacji produkcyjnej.

Ćwiczenie 1 — pełny cykl instalacji z weryfikacją kodu wyjścia

```
$proces = Start-Process msiexec.exe -ArgumentList '/i', '"C:\Sciezka\Pakiet.msi"', '/qn',
'/norestart' -Wait -PassThru
switch ($proces.ExitCode) {
    0    { "Sukces" }
    3010 { "Sukces, wymagany restart" }
    default { "Błąd, kod: $($proces.ExitCode)" }
}
```

Uruchamiamy instalację dowolnego dostępnego pakietu MSI testowego i sprawdzamy pełną obsługę trzech możliwych ścieżek wyniku.

Odpowiedź

`switch` z blokiem `default` jest tu czytelniejszy niż seria `if`/`elseif`/`else` przy więcej niż dwóch rozróżnianych wartościach — każdy kod wyjścia inny niż `0` i `3010` trafia do gałęzi domyślnej, gdzie warto dopisać dalsze rozgałęzienie na konkretne, znane kody błędów Windows Installer, jeśli skrypt ma je rozróżniać precyzyjniej niż samo „coś poszło nie tak".

Ćwiczenie 2 — porównanie dwóch mechanizmów komponentów

```
Get-WindowsCapability -Online -Name "*OpenSSH*"
Get-WindowsOptionalFeature -Online -FeatureName "*SSH*"
```

Sprawdzamy, w którym z dwóch mechanizmów faktycznie leży OpenSSH na naszym systemie.

Odpowiedź

OpenSSH pojawia się jako wynik `Get-WindowsCapability`, nie `Get-WindowsOptionalFeature` — potwierdza to, że jest zaimplementowany jako komponent (`Capability`), pobierany na żądanie, a nie jako starsza funkcja opcjonalna wbudowana od zawsze w obraz systemu. To praktyczna demonstracja rozróżnienia z sekcji „Jak to działa" wyżej.

Pytania kontrolne

1. Dlaczego `Get-Package` jest bezpieczniejszym wyborem do rutynowego sprawdzania zainstalowanego oprogramowania niż `Get-CimInstance Win32_Product`?

Odpowiedź

`Get-CimInstance Win32_Product` przy każdym wywołaniu wymusza ciche przeliczenie konfiguracji (`reconfiguration`) wszystkich zainstalowanych pakietów MSI na maszynie — udokumentowane przez Microsoft zachowanie, które potrafi zauważalnie obciążyć system i generować dodatkowe zdarzenia w dzienniku, szczególnie przy dużej liczbie zainstalowanych pakietów. `Get-Package` odczytuje te same informacje bez wywoływania tego efektu ubocznego.

2. Skrypt wdrożeniowy instaluje pięć pakietów MSI po kolei w pętli, każdy z /qn ale bez /norestart. Co może pójść nie tak?

Odpowiedź

Instalator jednego z pakietów w środku pętli może zażądać restartu systemu natychmiast po swoim zakończeniu, jeśli sam MSI ma taką flagę ustawioną — bez `/norestart` nic nie wstrzymuje tego restartu, co przerywa wykonanie skryptu w połowie sekwencji, zanim zdąży zainstalować pozostałe pakiety z listy. `/norestart` na każdym pakiecie odkłada wszystkie potencjalne restarty na jeden, świadomie zaplanowany moment po zakończeniu całej sekwencji.

Powiązane: Skrypty PowerShell: struktura i dobre praktyki · Skrypty audytowe Active Directory · WinRM: zarządzanie zdalne

Źródła: `Get-Help Start-Process` · `Get-Help Add-WindowsCapability` · dokumentacja Microsoft Learn, „Windows Installer error codes" i „winget command line reference"

Revision #41
Created 2026-05-08 13:59:49 UTC by Tomasz Konieczny
Updated 2026-10-23 17:14:13 UTC by Tomasz Konieczny