Sieć i zarządzanie zdalne

Konfiguracja sieci z wiersza poleceń, zapora, WinRM i poświadczenia.

- Sieć z wiersza poleceń
- WinRM: zarządzanie zdalne
- Poświadczenia i bezpieczne hasła w skryptach

Sieć z wiersza poleceń

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

Czas: ~30 min

Diagnostyka sieci z CLI — od samej listy adresów IP, przez firewall i sprawdzanie otwartych portów, po `Test-NetConnection`, które w praktyce zastąpiło osobne wywoływanie `ping`, `tracert` i `telnet`.

Skróty użyte na tej stronie (12)

Skrót	Co znaczy
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
ICMP	Internet Control Message Protocol: protokół diagnostyczny sieci, którym działa między innymi ping
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
IPv4	Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77
IPv6	Internet Protocol wersja 6: nowszy, dłuższy zapis adresów, w liczbach szesnastkowych oddzielonych dwukropkami
PID	Process ID: numer procesu w systemie
TCP	Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia
TTL	Time To Live: czas życia rekordu lub danych
UDP	User Datagram Protocol: szybki protokół sieciowy bez potwierdzeń
VPN	Virtual Private Network: szyfrowany tunel do sieci firmowej
WinRM	Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting

Fiszka

TYLKO ODCZYT

Sprawdzenie listy adresów IPv4 kart sieciowych: cmd z findstr

```
ipconfig /all | findstr /I "Adapter | Description | IPv4"
```

Filtrujemy surowe, wielolinijkowe wyjście `ipconfig /all` do samych linii zawierających nazwę karty, jej opis i adres IPv4 — `/I` w `findstr` włącza dopasowanie bez rozróżniania wielkości liter.

Przykład:

```
Ethernet adapter Ethernet:


   Description . . . . . . . . . . . : Intel(R) Ethernet Connection
   IPv4 Address. . . . . . . . . . . : 10.20.20.145(Preferred)
```

IPv4 = Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77 · **ipconfig** = IP configuration: konfiguracja sieci w Windowsie

TYLKO ODCZYT

Sprawdzenie listy adresów IPv4 kart sieciowych: PowerShell z ładnym formatowaniem

```
Get-NetAdapter | ForEach-Object {
    $adapterName = $_.InterfaceAlias
    $ipv4 = (Get-NetIPAddress -InterfaceAlias $adapterName -AddressFamily
IPv4).IPAddress
    [PSCustomObject]@{
        "Karta sieciowa" = $adapterName
        "Adres IPv4"     = $ipv4 -join ", "
    }
} | Format-Table -AutoSize
```

Budujemy własny, czytelny raport zamiast filtrować tekst wyjścia `ipconfig` — dla każdej karty sieciowej z `Get-NetAdapter` dociągamy jej adresy IPv4 przez `Get-NetIPAddress` i składamy w jeden obiekt na kartę.

Przykład:

```
Karta sieciowa      Adres IPv4
--------------      ----------
```

```
Ethernet            10.20.20.145
vEthernet (Default) 10.0.5.1
```

IPv4 = Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77

TYLKO ODCZYT

Wylistowanie wszystkich kart sieciowych: Get-NetAdapter

```
Get-NetAdapter | Format-Table Name, InterfaceDescription, Status, LinkSpeed -AutoSize
```

Zwracamy karty sieciowe wraz ze statusem łącza i negocjowaną prędkością — niezależnie od tego, czy mają przypisany adres IP.

Przykład:

```
Name        InterfaceDescription                    Status       LinkSpeed
----        --------------------                     ------       ---------
Ethernet    Intel(R) Ethernet Connection              Up            1 Gbps
Wi-Fi       Intel(R) Wireless-AC 9560                 Disconnected  0 bps
```

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)

TYLKO ODCZYT

Wyświetlenie pełnej konfiguracji IP interfejsu: Get-NetIPConfiguration

```
Get-NetIPConfiguration -InterfaceAlias "Ethernet"
```

Zwracamy w jednym poleceniu adres IP, bramę domyślną i serwery DNS przypisane do interfejsu — odpowiednik tego, co dawniej zbierało się z kilku osobnych wywołań `ipconfig /all` i ręcznego czytania.

Przykład:

```
InterfaceAlias       : Ethernet
InterfaceIndex       : 12
InterfaceDescription : Intel(R) Ethernet Connection
NetProfile.Name      : Sieć-firmowa
```

```
IPv4Address           : 10.20.20.145
IPv4DefaultGateway    : 10.20.20.1
DNSServer             : 10.20.10.11, 10.20.10.12
```

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **DNS** = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP

TYLKO ODCZYT

Sprawdzenie tablicy tras: Get-NetRoute

```
Get-NetRoute -AddressFamily IPv4 | Sort-Object RouteMetric | Select-Object
DestinationPrefix, NextHop, InterfaceAlias, RouteMetric
```

Zwracamy tablicę routingu z metrykami — przydatne, gdy serwer ma więcej niż jedną kartę sieciową i trzeba ustalić, którędy faktycznie pójdzie ruch do konkretnej podsieci.

Przykład:

```
DestinationPrefix NextHop    InterfaceAlias            RouteMetric
----------------- -------    --------------            -----------
0.0.0.0/0         10.20.20.1 Ethernet                           25
10.20.20.0/24     0.0.0.0    Ethernet                          281
127.0.0.0/8       0.0.0.0    Loopback Pseudo-Interface         331
```

TYLKO ODCZYT

Sprawdzenie łączności do hosta i portu: Test-NetConnection

```
Test-NetConnection -ComputerName AdresSerwera -Port NumerPortu
```

> **Pojęcie: Test-NetConnection.** Polecenie, które w PowerShellu zastąpiło osobne wywoływanie `ping` (test dostępności), `tracert` (trasa pakietu) i `telnet` (test otwartego portu TCP) — jedno polecenie zwraca zarówno wynik ICMP, jak i informację, czy wskazany port TCP odpowiada, bez potrzeby instalowania osobnego klienta Telnet tylko po to, żeby sprawdzić, czy port nasłuchuje.

Przykład:

```
ComputerName     : AdresSerwera
RemoteAddress    : 10.20.10.42
RemotePort       : 443
InterfaceAlias   : Ethernet
SourceAddress    : 10.20.20.145
TcpTestSucceeded : True
```

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia · **ICMP** = Internet Control Message Protocol: protokół diagnostyczny sieci, którym działa między innymi ping

TYLKO ODCZYT

Test-NetConnection z pełną diagnostyką trasy: -TraceRoute

```
Test-NetConnection -ComputerName AdresSerwera -TraceRoute
```

Dokładamy do zwykłego testu ICMP pełną trasę przeskoków między naszą maszyną a celem — odpowiednik `tracert`, wywołany bez opuszczania PowerShella.

ICMP = Internet Control Message Protocol: protokół diagnostyczny sieci, którym działa między innymi ping

TYLKO ODCZYT

Rozwiązanie nazwy DNS: Resolve-DnsName

```
Resolve-DnsName -Name NazwaDomeny -Type A
```

Zwracamy rekordy DNS bez wychodzenia poza PowerShella i bez zgadywania formatu wyniku `nslookup` — `Resolve-DnsName` zwraca obiekty z osobnymi właściwościami na typ rekordu, adres i TTL.

Przykład:

```
Name                                Type   TTL   Section    IPAddress
----                                ----   ---   -------    ---------
ad.zielona.example                  A      3600  Answer     10.20.10.11
```

DNS = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP · **TTL** = Time To Live: czas życia rekordu lub danych

□□ TYLKO ODCZYT

Wyczyszczenie lokalnego cache DNS: Clear-DnsClientCache

```
Clear-DnsClientCache
```

Usuwamy lokalnie zapamiętane wpisy DNS na bieżącej maszynie — potrzebne po zmianie rekordu na serwerze DNS, kiedy stara, nieaktualna odpowiedź wciąż jest podawana z pamięci podręcznej klienta, mimo że serwer DNS już dawno zwraca nową wartość.

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

□□ TYLKO ODCZYT

Zadanie, którego natywny PowerShell nie robi wprost: netsh

```
netsh interface ipv4 show interfaces
```

> **Pojęcie: netsh.** Starsze narzędzie konfiguracji sieci, sprzed ery PowerShella, wciąż obecne, bo część funkcji (szczegółowe ustawienia WinHTTP, konfiguracja proxy systemowego, eksport/import całej konfiguracji sieciowej jednym poleceniem) nie ma jeszcze pełnego odpowiednika w modułach `NetTCPIP` / `NetSecurity`. Sięgamy po nie punktowo, tam gdzie brakuje polecenia PowerShellowego, nie jako domyślne narzędzie do codziennej pracy.

Przykład:

```
Idx     Met         MTU          State                Name
---  ----------  ----------  ------------  ---------------------------
  1          75  4294967295  connected     Loopback Pseudo-Interface 1
 12          25        1500  connected     Ethernet
```

□□ **netsh** = network shell: konsola ustawień sieci Windowsa

□□ ZMIENIA SYSTEM

Wyeksportowanie całej konfiguracji sieciowej: netsh dump

```
netsh dump > C:\Backup\konfiguracja-sieci.txt
```

Zapisujemy pełną konfigurację interfejsów, firewalla i routingu w formacie, który to samo `netsh` potrafi później odtworzyć poleceniem `netsh exec` — przydatne jako migawka przed planowaną zmianą konfiguracji sieciowej serwera.

netsh = network shell: konsola ustawień sieci Windowsa

ZMIENIA SYSTEM

Dodanie reguł firewalla dla ECHO / PING / ICMP: New-NetFirewallRule

```
New-NetFirewallRule -DisplayName "PING / ECHO / ICMP v4" -Direction Inbound -Protocol
ICMPv4 -IcmpType 8 -Action Allow
New-NetFirewallRule -DisplayName "PING / ECHO / ICMP v6" -Direction Inbound -Protocol
ICMPv6 -IcmpType 8 -Action Allow
```

Otwieramy ruch przychodzący ICMP typu 8 (echo request) osobno dla IPv4 i IPv6 — domyślny profil zapory Windows blokuje odpowiadanie na ping z zewnątrz, więc te dwie reguły są jednym z pierwszych poleceń po świeżej instalacji serwera, do którego ktoś będzie chciał testować łączność pingiem.

ICMP = Internet Control Message Protocol: protokół diagnostyczny sieci, którym działa między innymi ping · **IPv4** = Internet Protocol wersja 4: adresy zapisane czterema liczbami z kropkami, na przykład 10.20.10.77 · **IPv6** = Internet Protocol wersja 6: nowszy, dłuższy zapis adresów, w liczbach szesnastkowych oddzielonych dwukropkami

OSTROŻNIE · NIEODWRACALNE

Usunięcie reguł ICMP: Remove-NetFirewallRule

```
Remove-NetFirewallRule -DisplayName "PING / ECHO / ICMP v4"
Remove-NetFirewallRule -DisplayName "PING / ECHO / ICMP v6"
```

Usuwamy regułę po jej nazwie wyświetlanej — `Remove-NetFirewallRule` wymaga dokładnego dopasowania `DisplayName`, w odróżnieniu od `Get-NetFirewallRule`, które akceptuje też wzorce z gwiazdką.

ICMP = Internet Control Message Protocol: protokół diagnostyczny sieci, którym działa między innymi ping

TYLKO ODCZYT

Sprawdzenie, czy reguła o danej nazwie istnieje: Get-NetFirewallRule z filtrem

```
Get-NetFirewallRule | Where-Object DisplayName -like "*PING*"
```

Szukamy reguł po fragmencie nazwy, zanim dodamy nową — zapobiega przypadkowemu utworzeniu dwóch reguł o niemal tej samej nazwie i funkcji.

Przykład:

```
Name                  DisplayName              Enabled  Direction  Action
----                  -----------              -------  ---------  ------
{a1b2c3d4-...}        PING / ECHO / ICMP v4    True     Inbound    Allow
{e5f6a7b8-...}        PING / ECHO / ICMP v6    True     Inbound    Allow
```

□□ TYLKO ODCZYT

Wylistowanie wszystkich reguł firewalla dla danego profilu

```
Get-NetFirewallRule -Direction Inbound -Enabled True | Where-Object Profile -like
"*Domain*" | Select-Object DisplayName, Action
```

> **Pojęcie: profil zapory.** Windows Firewall rozróżnia trzy profile — `Domain` (maszyna dołączona do domeny i widzi kontroler), `Private` (sieć oznaczona jako prywatna, np. domowa) i `Public` (sieć nieznana albo publiczna) — każda reguła jest przypisana do jednego, kilku albo wszystkich profili naraz, a obowiązujący w danej chwili zestaw reguł zależy od tego, jaki profil system rozpoznał dla aktywnego połączenia.

□□ ZMIENIA SYSTEM

Dodanie reguły firewalla dla konkretnego portu

```
New-NetFirewallRule -DisplayName "Aplikacja - port NumerPortu" -Direction Inbound -
Protocol TCP -LocalPort NumerPortu -Action Allow -Profile Domain
```

Otwieramy pojedynczy port TCP wyłącznie w profilu domenowym, zamiast we wszystkich trzech naraz — zawężenie profilu do minimum, jakiego dane połączenie realnie potrzebuje, jest dobrą praktyką niezależnie od tego, czy chodzi o port aplikacji, czy o WinRM opisany na stronie

WinRM: zarządzanie zdalne.

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia · **WinRM** = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting

ZMIENIA SYSTEM

Sprawdzenie otwartych / nasłuchujących portów wraz z aktywnym procesem: Get-NetTCPConnection

```
Get-NetTCPConnection |
    Sort-Object -Property LocalPort |
    Select-Object -Property LocalPort, RemoteAddress, RemotePort, State, OwningProcess |
    ForEach-Object {
        $_ | Add-Member -MemberType NoteProperty -Name ProcessName -Value (
            Get-Process -Id $_.OwningProcess -ErrorAction SilentlyContinue | Select-
Object -ExpandProperty ProcessName
        )
        $_
    } |
    Format-Table -Property LocalPort, RemoteAddress, RemotePort, State, ProcessName
```

`Get-NetTCPConnection` zwraca połączenia jako obiekty z gołym numerem `OwningProcess` — dociągamy nazwę procesu osobnym wywołaniem `Get-Process` i doklejamy ją jako nową właściwość przez `Add-Member`, żeby w jednej tabeli widzieć zarówno port, jak i to, co go faktycznie trzyma otwartym.

Przykład:

```
LocalPort RemoteAddress RemotePort State       ProcessName
--------- ------------- ---------- -----       -----------
       80 0.0.0.0               0 Listen      System
      443 0.0.0.0               0 Listen      w3wp
     3389 10.20.20.20       51422 Established svchost
     5985 0.0.0.0               0 Listen      svchost
```

ZMIENIA SYSTEM

Sprawdzenie połączeń filtrowanych po konkretnym adresie IP

```
$adresDoSprawdzenia = "AdresIP"

Get-NetTCPConnection | Where-Object {
    $_.RemoteAddress -eq $adresDoSprawdzenia -or $_.LocalAddress -eq $adresDoSprawdzenia
} |
    Sort-Object -Property LocalPort |
    Select-Object -Property LocalPort, RemoteAddress, RemotePort, State, OwningProcess |
    ForEach-Object {
        $_ | Add-Member -MemberType NoteProperty -Name ProcessName -Value (
            Get-Process -Id $_.OwningProcess -ErrorAction SilentlyContinue | Select-
Object -ExpandProperty ProcessName
        )
        $_
    } |
    Format-Table -Property LocalPort, RemoteAddress, RemotePort, State, ProcessName
```

Zawężamy ten sam raport do połączeń dotyczących jednego, interesującego nas adresu IP — przydatne przy śledzeniu, z czym konkretna maszyna komunikuje się przez sieć.

□□ **IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)

□□ **ZMIENIA SYSTEM**

Zapisanie listy połączeń do pliku CSV

```
$sciezkaCsv = "SciezkaDoPliku"

Get-NetTCPConnection |
    Sort-Object -Property LocalPort |
    Select-Object -Property LocalPort, RemoteAddress, RemotePort, State, OwningProcess |
    ForEach-Object {
        $_ | Add-Member -MemberType NoteProperty -Name ProcessName -Value (
            Get-Process -Id $_.OwningProcess -ErrorAction SilentlyContinue | Select-
Object -ExpandProperty ProcessName
        )
        $_
    } |
    Export-Csv -Path $sciezkaCsv -NoTypeInformation
```

```
Write-Host "Zapisano połączenia do pliku CSV: $sciezkaCsv"
```

Zapisujemy pełny raport połączeń do pliku zamiast wypisywać go na ekran — wygodne, gdy chcemy porównać stan portów w dwóch różnych momentach albo przekazać wynik komuś innemu bez zrzutu ekranu.

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

Sprawdzenie portów przez parsowanie netstat -ano

```
$netstatOutput = netstat -ano

$connections = @()
foreach ($line in $netstatOutput) {
    if ($line -match '^\s*(TCP|UDP)') {
        $parts = $line -split '\s+'
        $protocol = $parts[1]
        $localAddress = $parts[2]
        $remoteAddress = $parts[3]

        if ($protocol -eq 'TCP') {
            $state = $parts[4]
            $processId = $parts[5]
        } else {
            $state = "N/A"
            $processId = $parts[4]
        }

        function Split-AddressPort ($address) {
            if ($address -match '^\[(.+)\]:(\d+)$') {
                return @("[{0}]" -f $matches[1], $matches[2])
            } elseif ($address -match '^(.+):(\d+)$') {
                return @($matches[1], $matches[2])
            } else {
                return @($address, "N/A")
            }
```

```
        }

        $localIP, $localPort = Split-AddressPort $localAddress
        $remoteIP, $remotePort = Split-AddressPort $remoteAddress

        $processName = "Unknown"
        if ($processId -eq "0") {
            $processName = "Process terminated | PID = 0"
        } elseif ($processId -match '^\d+$') {
            $processName = (Get-Process -Id $processId -ErrorAction
SilentlyContinue).Name
        }

        $connections += [PSCustomObject]@{
            Protocol    = $protocol
            LocalIP     = $localIP
            LocalPort   = $localPort
            RemoteIP    = $remoteIP
            RemotePort  = $remotePort
            ProcessId   = $processId
            State       = $state
            ProcessName = $processName
        }
    }
}

$connections | Sort-Object -Property LocalIP | Format-Table -AutoSize
```

> **Pojęcie: netstat kontra Get-NetTCPConnection.** `netstat -ano` istnieje od czasów sprzed PowerShella i zwraca wyłącznie tekst — kolumny trzeba rozbijać ręcznie wyrażeniem regularnym, tak jak w powyższym skrypcie, licząc się z tym, że format kolumn (a zwłaszcza obsługa adresów IPv6 w nawiasach kwadratowych) różni się w szczegółach między wersjami Windowsa. `Get-NetTCPConnection` zwraca gotowe obiekty z osobnymi właściwościami i jest właściwym wyborem na nowo pisanym skrypcie — ten wariant zostawiamy jako wiedzę pomocniczą do czytania cudzych, starszych skryptów, które wciąż parsują `netstat` ręcznie.

IPv6 = Internet Protocol wersja 6: nowszy, dłuższy zapis adresów, w liczbach szesnastkowych oddzielonych dwukropkami · **TCP** = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia · **UDP** = User Datagram Protocol: szybki protokół sieciowy bez

potwierdzeń · **PID** = Process ID: numer procesu w systemie

TYLKO ODCZYT

Sprawdzenie portów w interaktywnej tabeli: Out-GridView

```
$netstatOutput = netstat -ano
$connections = @()

foreach ($line in $netstatOutput) {
    if ($line -match '^\s*(TCP|UDP)') {
        $parts = $line -split '\s+'
        $connections += [PSCustomObject]@{
            Protocol   = $parts[1]
            LocalPort  = $parts[2]
            RemoteAddr = $parts[3]
            ProcessId  = $parts[-1]
        }
    }
}

$connections | Sort-Object -Property LocalPort | Out-GridView -Title "Połączenia
TCP/UDP"
```

Ten sam pomysł, ale zamiast tabeli tekstowej otwieramy sortowalne, filtrowalne okno graficzne — wygodne przy jednorazowym, ręcznym przeglądaniu połączeń na stacji roboczej z interfejsem graficznym.

Uważaj

- `Get-NetTCPConnection` nie pokazuje połączeń UDP — dla nich potrzebne jest osobne `Get-NetUDPEndpoint`, albo parsowanie `netstat -ano` tak jak w wariancie wyżej.
- Reguła `New-NetFirewallRule` bez podanego `-Profile` obejmuje domyślnie wszystkie trzy profile naraz — warto zawężać świadomie, zwłaszcza przy portach administracyjnych.
- `Test-NetConnection` bez `-Port` wykonuje wyłącznie test ICMP — brak odpowiedzi nie zawsze oznacza, że host jest niedostępny, może oznaczać, że ICMP jest zablokowane firewallem, podczas gdy interesujący nas port TCP

działa normalnie.

UDP = User Datagram Protocol: szybki protokół sieciowy bez potwierdzeń · **ICMP** = Internet Control Message Protocol: protokół diagnostyczny sieci, którym działa między innymi ping · **TCP** = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

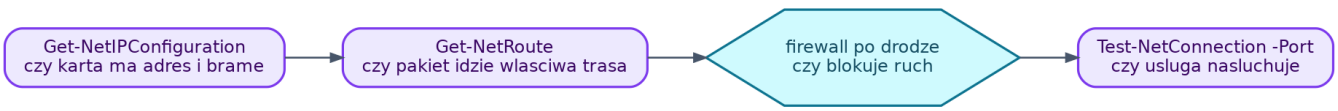
Z życia wzięte

Z ŻYCIA WZIĘTE

Kasia dzwoni z hotelu we wtorek wieczorem, bo VPN łączy się poprawnie, ale nie może dobić do rds01 pingiem, więc jest pewna, że tunel nie działa. Sprawdzamy `Test-NetConnection -ComputerName rds01 -Port 3389` i port odpowiada bez problemu, `TcpTestSucceeded : True`. Sam ping od dawna jest blokowany na profilu domenowym firewalla rds01, więc brak odpowiedzi na ICMP nie mówił nic o realnej dostępności pulpitu zdalnego.

Po co to istnieje

Diagnostyka sieciowa z CLI odpowiada na cztery pytania w kolejności, w jakiej zwykle się je zadaje przy awarii: czy nasza karta sieciowa w ogóle ma adres, czy pakiet wychodzi właściwą trasą, czy firewall po drugiej stronie (albo nasz własny) nie blokuje ruchu, i czy usługa na docelowej maszynie faktycznie nasłuchuje na oczekiwanym porcie. Historycznie każde z tych pytań wymagało osobnego narzędzia (`ipconfig`, `route print`, `netsh advfirewall`, `netstat`, `telnet`) o zupełnie różnej, tekstowej składni wyjścia. Moduły `NetTCPIP`, `NetSecurity` i `DnsClient` w PowerShellu zebrały te same odpowiedzi w jeden spójny zestaw poleceń zwracających obiekty, które da się dalej filtrować i sortować tym samym potokiem, którym operuje się na wszystkim innym w systemie.



Cztery pytania diagnostyki sieciowej po kolei: adres karty, trasa, firewall po drodze, port docelowy (kliknięcie otwiera pełny rozmiar)

Jak to działa

`Test-NetConnection` wewnętrznie wykonuje dokładnie te same operacje co osobne `ping`, `tracert` i test gniazda TCP — różnica jest w tym, że zwraca jeden obiekt z właściwościami `PingSucceeded`, `TcpTestSucceeded` i `TraceRoute` naraz, zamiast trzech osobnych, tekstowych wywołań do sparsowania ręcznie. To jest ten sam wzorzec, co przy `Get-NetTCPConnection` kontra `netstat` — nowsze polecenia PowerShellowe nie robią nic więcej niż stare narzędzia tekstowe, po prostu opakowują ten sam mechanizm systemowy w strukturę obiektową.

Częste błędy

Wnioskowanie z braku odpowiedzi na ping, że host jest całkowicie niedostępny. Domyślna konfiguracja Windows Firewall blokuje ICMP przychodzące — usługa może działać bezbłędnie na porcie TCP, a ping mimo to nie odpowie, dopóki nie dodamy reguł opisanych wyżej w tej fiszce.

Filtrowanie `Get-NetTCPConnection` po adresie IP jako tekście bez sprawdzenia formatu. Adresy IPv6 i skrócone zapisy `::` nie zawsze pasują do prostego porównania `-eq` z adresem wpisanym ręcznie — warto zweryfikować dokładny format zwracany przez `Get-NetTCPConnection`, zanim zbuduje się na nim automatyczny filtr.

Poleganie wyłącznie na `netstat -ano` w nowym skrypcie. Parsowanie tekstu wymaga wyrażeń regularnych podatnych na zmiany formatu między wersjami systemu — `Get-NetTCPConnection` daje ten sam wynik bez tego ryzyka i powinno być pierwszym wyborem, chyba że skrypt musi działać na systemie, gdzie moduł `NetTCPIP` z jakiegoś powodu nie jest dostępny.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell. Reguły firewalla wymagają uprawnień administratora — reszta ćwiczeń nie.

Ćwiczenie 1 — własny raport sieciowy

```
Get-NetIPConfiguration | Select-Object InterfaceAlias, IPv4Address, IPv4DefaultGateway,
DNSServer
Test-NetConnection -ComputerName 203.0.113.10 -Port 443
Resolve-DnsName -Name zielona.example -Type A | Select-Object -First 1
```

Zestawiamy trzy polecenia w kolejności, w jakiej zwykle diagnozuje się problem z łącznością: własna konfiguracja, dostępność zewnętrznego hosta na konkretnym porcie, rozwiązywanie nazw.

Odpowiedź

Jeśli pierwsze polecenie pokazuje poprawny adres i bramę, a drugie zwraca `TcpTestSucceeded : True`, problem z łącznością nie leży po stronie naszej karty sieciowej ani trasy do internetu. Jeśli trzecie polecenie nie zwraca żadnego adresu IP albo zwraca błąd, kolejnym krokiem jest sprawdzenie `Clear-DnsClientCache` i konfiguracji serwerów DNS z pierwszego polecenia, bo problem prawdopodobnie leży w rozwiązywaniu nazw, nie w samej łączności IP.

Ćwiczenie 2 — reguła firewalla i jej weryfikacja

```
New-NetFirewallRule -DisplayName "Test-Cwiczenie" -Direction Inbound -Protocol TCP -LocalPort 8080 -Action Allow
Get-NetFirewallRule -DisplayName "Test-Cwiczenie" | Get-NetFirewallPortFilter
Remove-NetFirewallRule -DisplayName "Test-Cwiczenie"
```

Tworzymy regułę, sprawdzamy, jaki port faktycznie objęła, i sprzątamy po sobie.

Odpowiedź

`Get-NetFirewallRule` zwraca samą regułę bez informacji o porcie — port, protokół i kierunek są osobnymi obiektami powiązanymi z regułą, dostępnymi przez osobne polecenia (`Get-NetFirewallPortFilter`, `Get-NetFirewallAddressFilter`). To odzwierciedla sposób, w jaki Windows Firewall przechowuje regułę wewnętrznie — jako zestaw powiązanych warunków, nie jeden płaski rekord.

Pytania kontrolne

1. Dlaczego `Get-NetTCPConnection | Select-Object OwningProcess` nie wystarcza, żeby dowiedzieć się, jaki program trzyma dany port, i trzeba dociągać `Get-Process` osobno?

Odpowiedź

`Get-NetTCPConnection` operuje na warstwie sieciowej systemu i zna wyłącznie identyfikator procesu (PID) trzymającego gniazdo — nie ma wbudowanego dostępu do tabeli procesów, którą zarządza inny podsystem. `Get-Process -Id` to osobne zapytanie do menedżera procesów, dlatego potrzebne jest jawne połączenie obu przez `ForEach-Object` i `Add-Member`, tak jak w fiszce wyżej.

2. Reguła ICMP jest dodana i włączona, a mimo to ping z zewnątrz wciąż nie działa. Jakie są dwa najbardziej prawdopodobne miejsca do sprawdzenia poza samą regułą?

Odpowiedź

Po pierwsze, profil zapory — reguła mogła zostać dodana tylko dla profilu `Domain`, podczas gdy aktywne połączenie sieciowe system rozpoznaje jako `Public`, więc reguła w ogóle nie obowiązuje dla bieżącego profilu. Po drugie, urządzenie sieciowe pomiędzy (router, firewall sprzętowy, grupa zabezpieczeń w chmurze) może blokować ICMP niezależnie od konfiguracji Windows Firewall na samej maszynie docelowej.

Powiązane: WinRM: zarządzanie zdalne · Podstawy składni PowerShella · Zadania zaplanowane, usługi i procesy

Źródła: `Get-Help Test-NetConnection` · `Get-Help New-NetFirewallRule` · dokumentacja Microsoft Learn, moduł „NetTCPIP" i „NetSecurity"

WinRM: zarządzanie zdalne

Poziom: średniozaawansowany

Dotyczy: Windows PowerShell 5.1, PowerShell 7+, Windows Server 2012 R2+

Czas: ~35 min

WinRM jest fundamentem, na którym stoi cały PowerShell Remoting — ta strona pokazuje, jak go uruchomić, jak się przez niego łączyć, jak wgrywać dane i skrypty na maszynę zdalną, i gdzie realnie kończy się jego bezpieczeństwo.

Skróty użyte na tej stronie (9)

Skrót	Co znaczy
CredSSP	Credential Security Support Provider: mechanizm Windows, który przekazuje poświadczenia na drugą maszynę; część poleceń klastra, na przykład tworzenie roli, wymaga go przy uruchamianiu zdalnym
DMTF	Distributed Management Task Force: organizacja, która opracowała standard Redfish
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
HTTP	Hypertext Transfer Protocol: protokół stron WWW
HTTPS	HTTP zabezpieczony szyfrowaniem TLS
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
RPC	Remote Procedure Call: sposób, w jaki programy wywołują funkcje na innym komputerze w sieci; kontrolery domeny używają go między innymi do replikacji
SMB	Server Message Block: protokół udostępniania plików i drukarek w sieci Windows, na nim stoi zwykły udział sieciowy
WinRM	Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting

Fiszka

ZMIENIA SYSTEM

Włączenie PowerShell Remoting na maszynie docelowej: Enable-PSRemoting

```
Enable-PSRemoting -Force
```

Uruchamiamy jako Administrator na maszynie, którą chcemy zarządzać zdalnie — polecenie startuje usługę WinRM, ustawia jej typ uruchomienia na automatyczny, tworzy nasłuch WinRM (`listener`) na domyślnym porcie i dodaje regułę firewalla dopuszczającą ruch przychodzący do WinRM w profilach domenowym i prywatnym. `-Force` pomija pytania potwierdzające, przydatne przy uruchamianiu w skrypcie bez nadzoru.

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

□□ **TYLKO ODCZYT**

Konfiguracja WinRM przez narzędzie klasyczne: winrm quickconfig

```
winrm quickconfig
```

Starszy, poprzedzający `Enable-PSRemoting` sposób osiągnięcia tego samego efektu — wciąż przydatny na maszynach, gdzie z jakiegoś powodu wolimy narzędzie wiersza poleceń zamiast cmdletu, albo gdzie skrypt musi działać identycznie na starszych wersjach Windowsa bez pełnego modułu `Microsoft.WSMan.Management`.

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

□□ **TYLKO ODCZYT**

Sprawdzenie, czy usługa WinRM działa

```
Get-Service WinRM
```

Weryfikujemy stan usługi przed próbą jakiegokolwiek połączenia zdalnego — bez uruchomionej usługi `WinRM` na maszynie docelowej każda próba `Enter-PSSession` albo `Invoke-Command` kończy się błędem połączenia, zanim dojdzie w ogóle do sprawdzenia poświadczeń.

Przykład:

```
Status   Name               DisplayName
------   ----               -----------
Running  WinRM              Windows Remote Management (WS-Management)
```

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

TYLKO ODCZYT

Przetestowanie dostępności WinRM na hoście zdalnym: Test-WSMan

```
Test-WSMan -ComputerName NazwaKomputera
```

Sprawdzamy z poziomu klienta, czy maszyna docelowa w ogóle odpowiada na protokół WS-Management, zanim spróbujemy nawiązać pełną sesję — szybszy i bardziej precyzyjny test niż `Test-NetConnection -Port 5985`, bo sprawdza sam protokół, nie tylko otwarty port.

Przykład:

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 10.0.20348 SP: 0.0 Stack: 3.0
```

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

ZMIENIA SYSTEM

Przepuszczenie ruchu sieciowego wymaganego przez WinRM

```
New-NetFirewallRule -DisplayName "Allow ICMPv4-In" -Protocol ICMPv4 -IcmpType 8 -
Direction Inbound -Action Allow -Profile Any
```

```
netsh advfirewall firewall add rule name="Open Port 139 - SMB over NetBIOS" dir=in
action=allow protocol=TCP localport=139
netsh advfirewall firewall add rule name="Open Port 445 - SMB" dir=in action=allow
protocol=TCP localport=445
```

Do zdalnego zarządzania systemem operacyjnym musi być uruchomiona usługa WinRM, ale w praktyce warto od razu przepuścić też ruch po portach 445 i 139 (SMB, wykorzystywany między innymi przy `Copy-Item -ToSession` i przy niektórych mechanizmach uwierzytelniania) oraz dodać regułę dla ping, żeby test łączności działał bez dodatkowej konfiguracji.

WinRM = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting · **SMB** = Server Message Block: protokół udostępniania plików i drukarek w sieci Windows, na nim stoi zwykły udział sieciowy · **netsh** = network shell: konsola ustawień sieci Windowsa · **TCP** = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

ZMIENIA SYSTEM

Rozwiązanie błędu Access Denied (5) po stronie docelowej

```
$newItemPropertySplat = @{
    Name         = 'LocalAccountTokenFilterPolicy'
    Path         = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System'
    PropertyType = 'DWord'
    Value        = 1
}
New-ItemProperty @newItemPropertySplat
```

Pojęcie: LocalAccountTokenFilterPolicy. Domyślnie Windows wydaje kontu lokalnemu (nie domenowemu) łączącemu się zdalnie ograniczony token — wystarczający do uwierzytelnienia, ale niewystarczający do wielu operacji administracyjnych, co objawia się błędem `Access Denied (5)` mimo poprawnego hasła i mimo że to samo konto ma pełne uprawnienia administratora lokalnie. Ustawienie tego klucza rejestru na `1` wyłącza filtrowanie tokenu dla kont lokalnych — stosujemy je świadomie, zwykle tylko tam, gdzie łączymy się kontem lokalnym administratora, a nie kontem domenowym.

HKLM = HKEY_LOCAL_MACHINE: gałąź rejestru Windows z ustawieniami całego komputera, a nie pojedynczego użytkownika

ZMIENIA SYSTEM

Dodanie hosta do zaufanych klientów: TrustedHosts

```
Set-Item WSMan:\localhost\Client\TrustedHosts -Value "NazwaKomputera" -Force
```

> **Pojęcie: TrustedHosts.** Lista hostów, do których klient WinRM zgadza się łączyć uwierzytelnianiem `Basic` albo bez weryfikacji certyfikatu Kerberos — potrzebna wtedy, gdy łączymy się z maszyną spoza domeny albo po adresie IP zamiast po nazwie, bo Kerberos (domyślny mechanizm uwierzytelniania WinRM w domenie) nie potrafi zweryfikować tożsamości hosta w takich warunkach. Wewnątrz jednej domeny, łącząc się po nazwie DNS znanej kontrolerowi domeny, `TrustedHosts` zwykle nie jest potrzebne wcale — Kerberos załatwia weryfikację sam. Wartość `*` dopuszcza dowolny host i jest uzasadniona wyłącznie na wydzielonej stacji administracyjnej w izolowanej sieci, nigdy na maszynie produkcyjnej dostępnej z zewnątrz.

WinRM = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting · **IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **DNS** = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP

TYLKO ODCZYT

Sprawdzenie bieżącej zawartości TrustedHosts

```
Get-Item WSMan:\localhost\Client\TrustedHosts
```

Przykład:

```
   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client


Type            Name                           SourceOfValue   Value
----            ----                           -------------   -----
System.String   TrustedHosts                                  NazwaKomputera
```

TYLKO ODCZYT

Podłączanie się do WinRM z PowerShella: Enter-PSSession

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

Uruchamiamy z poziomu PowerShella, po uruchomieniu usługi WinRM na maszynie docelowej — otwieramy interaktywną sesję zdalną, w której każde kolejne polecenie wykonuje się fizycznie na maszynie docelowej, nie lokalnie, dopóki nie wpiszemy `Exit-PSSession` albo `exit` .

Przykład:

```
[NazwaKomputera]: PS C:\Users\nazwa_usera\Documents>
```

Zmiana znaku zachęty na `[NazwaKomputera]:` jest jedynym wizualnym sygnałem, że polecenia wykonują się już zdalnie — łatwo o tym zapomnieć w środku dłuższej sesji.

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

TYLKO ODCZYT

Zdalne uruchamianie skryptów PS: Invoke-Command ze ScriptBlock

```
Invoke-Command -ComputerName NazwaKomputera -ScriptBlock {
    PowerShell.exe -ExecutionPolicy Bypass -File "SciezkaDoPliku"
} -Credential (Get-Credential)
```

Wykonujemy z poziomu PowerShella jako użytkownik z uprawnieniami administracyjnymi na komputerze docelowym — w odróżnieniu od `Enter-PSSession`, `Invoke-Command` nie otwiera interaktywnej sesji, tylko wykonuje podany blok skryptu i zwraca wynik do sesji lokalnej, co nadaje się do uruchamiania na wielu maszynach naraz bez ręcznego wchodzenia na każdą z osobna.

PS = PowerShell: powłoka i język skryptowy Windowsa (w Linuksie ps to polecenie process status)

TYLKO ODCZYT

Zdalne uruchamianie lokalnego pliku skryptu: Invoke-Command -FilePath

```
Invoke-Command -ComputerName NazwaKomputera -FilePath "SciezkaDoPliku" -Credential (Get-Credential)
```

Zamiast wklejać treść skryptu w bloku `-ScriptBlock`, wskazujemy plik `.ps1` leżący lokalnie — `Invoke-Command` sam przesyła jego treść na maszynę docelową i tam ją wykonuje, więc plik nie musi wcześniej istnieć fizycznie po drugiej stronie.

TYLKO ODCZYT

Uruchomienie tego samego polecenia na wielu maszynach naraz

```
Invoke-Command -ComputerName "NazwaKomputera1", "NazwaKomputera2", "NazwaKomputera3" -
ScriptBlock {
    Get-Service -Name wuauserv | Select-Object PSComputerName, Status
} -Credential (Get-Credential)
```

`-ComputerName` przyjmuje listę hostów naraz — `Invoke-Command` łączy się z każdym z nich równolegle i zwraca wyniki oznaczone właściwością `PSComputerName`, dzięki czemu od razu widać, z której maszyny pochodzi który wiersz wyniku.

◫◫ ZMIENIA SYSTEM

Utworzenie trwałej sesji zdalnej: New-PSSession

```
$sesja = New-PSSession -ComputerName NazwaKomputera -Credential (Get-Credential)
```

> **Pojęcie: sesja trwała.** `Enter-PSSession` i zwykłe `Invoke-Command` otwierają połączenie, wykonują polecenie i zamykają je zaraz potem — `New-PSSession` tworzy połączenie, które zostaje otwarte i przechowuje stan (zmienne, zaimportowane moduły, bieżący katalog) między kolejnymi wywołaniami, dopóki jawnie go nie zamkniemy poleceniem `Remove-PSSession`. Przydatne, gdy trzeba wykonać serię powiązanych operacji na tej samej maszynie bez każdorazowego ponownego uwierzytelniania.

◫◫ TYLKO ODCZYT

Wykonanie polecenia w istniejącej sesji trwałej

```
Invoke-Command -Session $sesja -ScriptBlock { Get-ChildItem C:\Temp }
```

Odwołujemy się do wcześniej utworzonej zmiennej sesji zamiast podawać `-ComputerName` na nowo — zmienne ustawione w poprzednim wywołaniu w tej samej sesji są wciąż dostępne.

◫◫ ZMIENIA SYSTEM

Wgranie danych poprzez WinRM: Copy-Item -ToSession

```
$cred = Get-Credential
$session = New-PSSession -ComputerName NazwaKomputera -Credential $cred
```

```
Copy-Item "SciezkaZrodlowa" `
    -Destination "SciezkaDocelowa" `
    -ToSession $session
```

Wcześniej musi być uruchomiona usługa WinRM oraz musimy mieć włączony dostęp sieciowy do danego komputera. `Copy-Item` z `-ToSession` przesyła plik z maszyny lokalnej na maszynę zdalną przez ten sam kanał WinRM, którym działa cała reszta PowerShell Remoting — bez potrzeby udostępniania osobnego udziału SMB tylko po to, żeby skopiować jeden plik.

WinRM = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting · **SMB** = Server Message Block: protokół udostępniania plików i drukarek w sieci Windows, na nim stoi zwykły udział sieciowy

TYLKO ODCZYT

Pobranie danych z maszyny zdalnej: Copy-Item -FromSession

```
Copy-Item -Path "SciezkaNaMaszynieZdalnej" -Destination "SciezkaLokalna" -FromSession
$session
```

Kierunek odwrotny do powyższego — ściągamy plik z maszyny zdalnej na lokalną, korzystając z tej samej, już otwartej sesji.

ZMIENIA SYSTEM

Weryfikacja poświadczeń lokalnych użytkownika poprzez WinRM

```
$u = Read-Host 'Login (bez domeny)'
$p = Read-Host "Haslo dla $u" -AsSecureString
$pl =
[Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.InteropServices.Marshal]::Se
cureStringToBSTR($p))
Add-Type -AssemblyName System.DirectoryServices.AccountManagement
$ok =
[System.DirectoryServices.AccountManagement.PrincipalContext]::new('Machine').ValidateCr
edentials($u, $pl)

Write-Host ''
if ($ok) {
    Write-Host '  [ OK ] ' -ForegroundColor Black -BackgroundColor Green -NoNewline
```

```
    Write-Host " $env:COMPUTERNAME\$u " -ForegroundColor Green
    Write-Host '  Poswiadczenia poprawne.' -ForegroundColor DarkGray
} else {
    Write-Host '  [ FAIL ] ' -ForegroundColor White -BackgroundColor Red -NoNewline
    Write-Host " $env:COMPUTERNAME\$u " -ForegroundColor Red
    Write-Host '  Zle haslo / konto nie istnieje / zablokowane.' -ForegroundColor
DarkGray
}
Write-Host ''
```

Przed wykonaniem skryptu łączymy się zdalnie do maszyny, na której chcemy sprawdzić poświadczenia (przez `Enter-PSSession` albo `Invoke-Command`), i dopiero wewnątrz tej sesji wykonujemy powyższy kod. Skrypt sprawdza login i hasło konta lokalnego wprost przez `PrincipalContext` z kontekstem `Machine`, bez logowania interaktywnego — przydatne przy diagnozowaniu zgłoszeń „nie mogę się zalogować", gdzie trzeba potwierdzić, czy problem leży w haśle, czy gdzie indziej, bez ryzyka zablokowania konta powtarzanymi próbami logowania na pulpicie.

Przykład:

```
  [ OK ]  RDS01\halina.kowalska
  Poswiadczenia poprawne.
```

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

ZMIENIA SYSTEM

Zmiana pliku hosts na potrzeby połączenia się do WinRM przez inny adres IP

```
$plikHosts = 'C:\Windows\System32\drivers\etc\hosts'
$wpis = 'AdresIP NazwaDNS'

Get-Content $plikHosts
Copy-Item $plikHosts "$plikHosts.bak" -Force

try {
    Add-Content $plikHosts "`n$wpis" -Encoding ASCII
    Clear-DnsClientCache
```

```
    $s = New-PSSession -ComputerName NazwaDNS -Credential $cred -ErrorAction Stop
}
finally {
    Copy-Item "$plikHosts.bak" $plikHosts -Force
    Clear-DnsClientCache
}

Enter-PSSession -Session $s
```

Uruchamiamy PowerShell jako administrator. Sytuacja: musimy połączyć się do WinRM po nazwie DNS, którą Kerberos zweryfikuje poprawnie, ale rozwiązanie DNS tej nazwy w danej chwili prowadzi do niewłaściwego adresu (albo w ogóle nie istnieje) — dopisujemy tymczasowo wpis w pliku `hosts`, wymuszający właściwy adres IP dla tej nazwy, nawiązujemy sesję, a w bloku `finally` przywracamy oryginalny plik `hosts` z kopii zapasowej niezależnie od tego, czy połączenie się powiodło. Blok `try/finally` gwarantuje, że tymczasowy wpis nigdy nie zostanie w pliku `hosts` na stałe, nawet jeśli `New-PSSession` zawiedzie.

WinRM = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting · **IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **DNS** = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP · **ASCII** = American Standard Code for Information Interchange: podstawowy zestaw znaków (litery, cyfry) używany w komputerach

TYLKO ODCZYT

HTTP kontra HTTPS w WinRM

Wariant	Port domyślny	Szyfrowanie transportu
WinRM przez HTTP	`5985`	ruch szyfrowany na poziomie komunikatów (Kerberos/NTLM), nie na poziomie transportu
WinRM przez HTTPS	`5986`	pełne szyfrowanie TLS, wymaga certyfikatu na nasłuchu

Domyślna konfiguracja `Enable-PSRemoting` tworzy nasłuch na porcie 5985 (HTTP) — bezpieczne wewnątrz domeny, gdzie Kerberos i tak szyfruje treść komunikatów, ale niewystarczające, jeśli sesja ma przechodzić przez sieć nieszyfrowaną poza granicą zaufanej infrastruktury. Nasłuch HTTPS na porcie 5986 wymaga wcześniej wystawionego certyfikatu z nazwą pasującą do nazwy hosta i jest konfigurowany osobno poleceniem `New-Item` w dostawcy `WSMan:`.



Klient laczy sie z serwerem WinRM przez ustalony port: 5985 HTTP albo 5986 HTTPS (kliknięcie otwiera pełny rozmiar)

Uważaj

- `TrustedHosts` ustawione na `*` na maszynie z dostępem do internetu jest realnym ryzykiem — klient zaakceptuje uwierzytelnianie `Basic` do dowolnego hosta, łącznie z takim, który podszywa się pod cel połączenia.
- `LocalAccountTokenFilterPolicy` ustawione na `1` obniża izolację kont lokalnych na całej maszynie, nie tylko dla jednego połączenia — stosujemy je świadomie i tam, gdzie to naprawdę konieczne.
- Zmiana pliku `hosts` bez bloku `try/finally` zostawia tymczasowy wpis na stałe, jeśli coś pójdzie nie tak w trakcie — kolejna osoba diagnozująca problem z rozwiązywaniem nazw traci na tym godziny.

HTTP = Hypertext Transfer Protocol: protokół stron WWW · **HTTPS** = HTTP zabezpieczony szyfrowaniem TLS · **WinRM** = Windows Remote Management: zdalne zarządzanie Windowsem, podstawa PowerShell Remoting · **TLS** = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

Z życia wzięte

Z ŻYCIA WZIĘTE

Tomek próbuje w piątek o 16:00 połączyć się z rds01 przez `Enter-PSSession`, ale dostaje błąd `WinRM cannot process the request`, mimo że godzinę wcześniej działało to bez problemu z innej stacji. Sprawdza `Test-WSMan -ComputerName rds01` i dostaje tę samą odpowiedź co zawsze, więc usługa na pewno działa. Problem okazuje się banalny: tym razem łączył się po adresie IP zamiast po nazwie DNS, a Kerberos nie potrafi zweryfikować tożsamości hosta

> podanego jako sam adres.

Po co to istnieje

> **Pojęcie: WS-Management.** Otwarty, oparty na SOAP protokół firmowany przez DMTF do zdalnego zarządzania sprzętem i oprogramowaniem niezależnie od producenta — WinRM (Windows Remote Management) to implementacja tego protokołu na Windowsie, a PowerShell Remoting to warstwa PowerShella zbudowana na WinRM, pozwalająca przesyłać przez ten sam kanał nie tylko pojedyncze polecenia, ale całe sesje z zachowanym stanem. Zanim WinRM stał się standardem, zdalne zarządzanie Windowsem opierało się głównie na DCOM/WMI (RPC po porcie 135 i dynamicznych portach powyżej) — trudnym do przepuszczenia przez firewalle segmentujące sieć, bo zestaw potrzebnych portów nie był z góry znany. WinRM komunikuje się jednym, ustalonym portem (5985 albo 5986), co czyni go znacznie łatwiejszym do kontrolowanego otwarcia w regułach firewalla.

Analogia

WinRM działa jak recepcja w budynku biurowym z kontrolą dostępu, a nie jak wspólny hol bez recepcjonisty. Zanim ktokolwiek wejdzie do środka i zacznie coś robić na dowolnym piętrze (czyli wykona polecenie na maszynie docelowej), musi przejść przez jedno, ustalone wejście (port 5985 albo 5986), pokazać przepustkę (poświadczenia) i dopiero recepcja decyduje, czy i dokąd go wpuścić. Stary model DCOM/WMI przypominał budynek z wieloma niepilnowanymi wejściami na różnych ścianach, których numeru nie dało się z góry przewidzieć — trudny do ochrony, bo firewall musiałby pilnować wszystkich naraz. Ograniczenie tej analogii: sama recepcja (WinRM) weryfikuje tylko, kto wchodzi — nie decyduje sama z siebie, co dana osoba może zrobić w środku, tym zajmują się uprawnienia konta na maszynie docelowej, dokładnie tak samo jak bez WinRM.

Jak to działa

Mechanizm `TrustedHosts` istnieje, bo domyślne uwierzytelnianie WinRM wewnątrz domeny opiera się na Kerberosie, który weryfikuje tożsamość zarówno klienta, jak i serwera przez kontroler domeny — łącząc się po adresie IP albo z hosta spoza domeny, Kerberos nie ma jak wykonać tej weryfikacji, więc WinRM domyślnie odmawia połączenia, chyba że jawnie powiemy klientowi „ufaj temu hostowi mimo braku weryfikacji Kerberosa". To dlatego `TrustedHosts` jest ustawieniem po stronie klienta, nie serwera — to klient decyduje, komu ufa bez pełnej weryfikacji tożsamości drugiej strony.

Podwójny przeskok (double hop). Gdy łączymy się przez WinRM na maszynę A (`Enter-PSSession -ComputerName A`), a stamtąd próbujemy sięgnąć po zasób na maszynie B wymagający uwierzytelnienia (na przykład udział sieciowy albo baza danych), domyślne poświadczenia Kerberos nie „podróżują" automatycznie z sesji lokalnej przez maszynę A do maszyny B — to ograniczenie bezpieczeństwa Kerberosa, zwane właśnie podwójnym przeskokiem, chroniące przed przejęciem tokenu użytkownika przez pośredniczącą maszynę. CredSSP (Credential Security Support Provider) jest jednym ze sposobów obejścia tego ograniczenia — przekazuje poświadczenia w postaci umożliwiającej ich dalsze użycie na maszynie A tak, jakby to A logowało się nimi samodzielnie do B. Włączenie CredSSP (`Enable-WSManCredSSP`) wymaga świadomej decyzji, bo maszyna pośrednicząca dostaje wtedy realną możliwość użycia naszych poświadczeń do czegokolwiek, do czego akurat ma sieciowy dostęp — stosujemy je wyłącznie tam, gdzie podwójny przeskok jest naprawdę niezbędny, nigdy jako domyślne ustawienie.

Częste błędy

Błąd `Access Denied (5)` mimo poprawnego hasła konta lokalnego. Najczęstsza przyczyna to domyślne filtrowanie tokenu opisane w fiszce `LocalAccountTokenFilterPolicy` wyżej — konto domenowe tego problemu zwykle nie ma, bo nie podlega temu samemu mechanizmowi filtrowania.

Błąd `WinRM cannot process the request` przy próbie połączenia po adresie IP. Kerberos nie weryfikuje adresów IP jako tożsamości hosta — trzeba albo dodać host do `TrustedHosts` po stronie klienta, albo połączyć się po nazwie DNS zamiast po adresie IP.

Zapomnienie o zamknięciu sesji trwałej. `New-PSSession` bez późniejszego `Remove-PSSession` zostawia otwarte połączenie po stronie serwera aż do wygaśnięcia limitu czasu — przy dużej liczbie skryptów tworzących sesje bez sprzątania po sobie serwer WinRM potrafi wyczerpać domyślny limit jednoczesnych sesji (`MaxShellsPerUser`).

□□ Ćwiczenia

> Ćwiczenia poniżej wymagają dwóch maszyn z Windowsem w tej samej sieci testowej (albo dwóch maszyn wirtualnych) i uprawnień administratora na obu.

Ćwiczenie 1 — od zera do pierwszej sesji

```
# Na maszynie docelowej
Enable-PSRemoting -Force
Get-Service WinRM
```

```
# Na maszynie klienckiej
Test-WSMan -ComputerName NazwaKomputeraDocelowego
Enter-PSSession -ComputerName NazwaKomputeraDocelowego -Credential (Get-Credential)
```

Przechodzimy przez pełną ścieżkę: włączenie WinRM po stronie serwera, weryfikacja z klienta, nawiązanie sesji.

Odpowiedź

Jeśli obie maszyny są w tej samej domenie i łączymy się po nazwie DNS, `TrustedHosts` po stronie klienta nie jest potrzebne — Kerberos weryfikuje tożsamość automatycznie. Jeśli maszyny są w grupie roboczej (bez domeny), `Enter-PSSession` zwróci błąd uwierzytelniania, dopóki nie dodamy hosta docelowego do `TrustedHosts` klienta, bo bez domeny nie ma Kerberosa, który mógłby zweryfikować tożsamość inaczej.

Ćwiczenie 2 — sesja trwała kontra Invoke-Command

```
$sesja = New-PSSession -ComputerName NazwaKomputera -Credential (Get-Credential)
Invoke-Command -Session $sesja -ScriptBlock { $test = "wartosc-testowa" }
Invoke-Command -Session $sesja -ScriptBlock { $test }
Invoke-Command -ComputerName NazwaKomputera -ScriptBlock { $test } -Credential (Get-
Credential)
Remove-PSSession $sesja
```

Dlaczego trzecie polecenie nie zwraca `wartosc-testowa`, mimo że ustawiliśmy tę zmienną chwilę wcześniej na tej samej maszynie?

Odpowiedź

Zwykłe `Invoke-Command -ComputerName` bez `-Session` otwiera nowe, tymczasowe połączenie, wykonuje polecenie i od razu je zamyka — nie ma dostępu do stanu (zmiennych) utworzonego w poprzedniej, osobnej sesji `$sesja`. Tylko odwołanie się do tej samej, wciąż otwartej sesji przez `-Session $sesja` daje dostęp do zmiennych ustawionych wcześniej w jej kontekście, co pokazuje drugie polecenie w tym ćwiczeniu.

Pytania kontrolne

1. Dlaczego dodanie hosta do TrustedHosts po stronie klienta nie rozwiązuje problemu podwójnego przeskoku (double hop)?

Odpowiedź

`TrustedHosts` dotyczy wyłącznie tego, czy klient zgadza się nawiązać pierwsze połączenie z danym hostem bez pełnej weryfikacji Kerberosa — nie ma nic wspólnego z tym, czy poświadczenia użyte do tego połączenia mogą zostać dalej przekazane z maszyny pośredniczącej do trzeciej maszyny. To dwa zupełnie osobne mechanizmy: jeden kontroluje zaufanie do pierwszego przeskoku, drugi (CredSSP albo delegowanie Kerberos) kontroluje, czy poświadczenia „podróżują" dalej.

2. Skrypt zmieniający plik hosts zawiera blok try/finally przywracający oryginalny plik. Co konkretnie by się stało, gdyby ten blok finally usunąć, a New-PSSession zawiodłoby z błędem w środku bloku try?

Odpowiedź

Bez `finally` tymczasowy wpis dodany do pliku `hosts` zostałby na stałe, mimo że sesja się nie udała — kolejna próba rozwiązania tej samej nazwy DNS na tej maszynie zwracałaby błędny, ręcznie wymuszony adres IP, co przy diagnozowaniu późniejszych problemów z siecią byłoby mylące i trudne do skojarzenia z tym jednorazowym eksperymentem sprzed tygodni. `finally` gwarantuje wykonanie sprzątania niezależnie od tego, czy blok `try` zakończył się sukcesem, czy wyjątkiem.

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

Źródła: `Get-Help about_Remote` · `Get-Help about_WSMan_Provider` · `Get-Help Enable-PSRemoting` · dokumentacja Microsoft Learn, „WS-Management protocol"

Poświadczenia i bezpieczne hasła w skryptach

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

Czas: ~25 min

Zaszyfrowanie hasła w skrypcie nie jest trudne — trudne jest zrozumienie, dla kogo dokładnie to hasło pozostanie czytelne, a dla kogo nie. Ta pułapka wraca w każdym audycie skryptów administracyjnych.

Skróty użyte na tej stronie (6)

Skrót	Co znaczy
AES	Advanced Encryption Standard: powszechny szyfr blokowy, którym szyfrują dane karty DESFire i wiele innych systemów, w naszych przykładach z kluczem 128 bitów
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
DPAPI	Data Protection API: mechanizm Windowsa szyfrujący dane kluczem konkretnego użytkownika
gMSA	group Managed Service Account: konto usługi, któremu Windows sam zmienia hasło
GUI	Graphical User Interface: okienka i myszka zamiast komend
IT	Information Technology: dział albo praca związana z komputerami i infrastrukturą firmy

Fiszka

TYLKO ODCZYT

Interaktywne pobranie poświadczeń: Get-Credential

```
$poswiadczenia = Get-Credential -Message "Podaj poświadczenia do domeny NazwaDomeny"
```

> **Pojęcie: PSCredential.** `Get-Credential` otwiera okno (albo, w sesji bez GUI, pyta w konsoli) o login i hasło i zwraca obiekt typu `PSCredential` — parę: nazwę użytkownika jako zwykły tekst i hasło jako `SecureString`, nigdy jako gołe hasło w pamięci przez dłużej niż to konieczne do jego zaszyfrowania. To jest standardowy typ, jakiego oczekują wszystkie polecenia PowerShella przyjmujące parametr `-Credential` — `Enter-PSSession`, `Invoke-Command`, `Get-ADUser` i dziesiątki innych.

Przykład:

```
UserName                    Password
--------                    --------
NAZWADOMENY\nazwa_usera     System.Security.SecureString
```

GUI = Graphical User Interface: okienka i myszka zamiast komend

ZMIENIA SYSTEM

Zbudowanie obiektu PSCredential bez okna dialogowego

```
$uzytkownik = "NazwaDomeny\nazwa_usera"
$hasloBezpieczne = ConvertTo-SecureString "HasloDoUzycia" -AsPlainText -Force
$poswiadczenia = New-Object System.Management.Automation.PSCredential ($uzytkownik,
$hasloBezpieczne)
```

Budujemy `PSCredential` programowo, na przykład na potrzeby skryptu nienadzorowanego, gdzie nikt nie odpowie na okno dialogowe `Get-Credential`. Hasło w postaci zwykłego tekstu (`-AsPlainText`) istnieje w kodzie tylko chwilę, do momentu przekonwertowania na `SecureString` — jeśli to hasło wpisujemy wprost w skrypcie zamiast wczytywać z zaszyfrowanego pliku, wracamy do problemu hasła jawnego w kodzie źródłowym, który cała reszta tej strony ma rozwiązać.

ZMIENIA SYSTEM

Zaszyfrowanie hasła dla secure-string w PowerShell

```
'HasloDoZaszyfrowania' | ConvertTo-SecureString -AsPlainText -Force | ConvertFrom-
SecureString
```

Komendę uruchamiamy jako użytkownik, który będzie z tego hasła korzystał. Hasło wygenerowane w ten sposób będzie działać poprawnie tylko i wyłącznie z konta, z którego

wykonamy tę komendę — jeżeli PowerShell uruchomimy jako administrator, kontem wykonawczym, na przykład w harmonogramie zadań, musi być administrator, inaczej odszyfrowanie zawiedzie mimo poprawnie przepisanego ciągu znaków.

Przykład:

```
01000000d08c9ddf0115d1118c7a00c04fc297eb0100000004a9421370950646 90f68641a776fe4000000000
200000000001066000
0c00000010000000921ea8a928f125ebe0bb3b5dfb7445b1000000004800000a00000001000000c63ba1a67a
96b5fe7f64b9265e8d
35757fc469a95686571
3779ab180e276931a4f9af82e7e32b3d6ce9573bd2c965075a24e914000000c6dd47ea70d603669879f2023
4abc33a1cbb4818
```

Pamiętamy o usunięciu wszystkich znaków nowej linii z zapisanego ciągu przed jego ponownym wczytaniem — kopiowanie z okna terminala czasem dokleja twarde zawijanie wierszy, którego nie ma w oryginalnym wyniku.

◫◫ ZMIENIA SYSTEM

Odczytanie zaszyfrowanego hasła z powrotem do PSCredential

```
$hasloZaszyfrowane = Get-Content "SciezkaDoPliku.txt"
$uzytkownik = "NazwaDomeny\nazwa_usera"
$poswiadczenia = New-Object System.Management.Automation.PSCredential (
    $uzytkownik,
    ($hasloZaszyfrowane | ConvertTo-SecureString)
)
```

`ConvertTo-SecureString` bez `-AsPlainText`, wywołane na już zaszyfrowanym ciągu, odszyfrowuje go z powrotem do postaci `SecureString` w pamięci — pod warunkiem że wykonuje to ten sam użytkownik, na tej samej maszynie, która wygenerowała oryginalny ciąg.

◫◫ TYLKO ODCZYT

Odszyfrowanie SecureString do zwykłego tekstu, gdy naprawdę to konieczne

```
$tekstJawny = [Runtime.InteropServices.Marshal]::PtrToStringAuto(
    [Runtime.InteropServices.Marshal]::SecureStringToBSTR($poswiadczenia.Password)
)
```

Odwracamy `SecureString` z powrotem do zwykłego tekstu — potrzebne tam, gdzie docelowe API (na przykład stare biblioteki .NET albo wywołanie zewnętrznego programu przez `Start-Process -ArgumentList`) nie przyjmuje `SecureString` wprost i wymaga gołego ciągu znaków. Robimy to świadomie, na chwilę, tuż przed użyciem — nie przechowujemy wyniku dłużej niż to konieczne, bo od tego momentu hasło znowu istnieje w pamięci procesu jako zwykły tekst.

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą

ZMIENIA SYSTEM

Wygenerowanie własnego klucza AES do przenośnego szyfrowania

```
$kluczAES = New-Object Byte[] 32
[Security.Cryptography.RNGCryptoServiceProvider]::Create().GetBytes($kluczAES)
$kluczAES | Out-File "SciezkaDoKlucza.key"
```

Generujemy losowy klucz 256-bitowy i zapisujemy go do osobnego pliku — ten plik trzeba potem chronić co najmniej tak samo starannie jak samo hasło, bo staje się jedynym sekretem potrzebnym do odszyfrowania wszystkiego, co nim zaszyfrujemy.

AES = Advanced Encryption Standard: powszechny szyfr blokowy, którym szyfrują dane karty DESFire i wiele innych systemów, w naszych przykładach z kluczem 128 bitów

ZMIENIA SYSTEM

Zaszyfrowanie hasła własnym kluczem AES zamiast DPAPI

```
$kluczAES = Get-Content "SciezkaDoKlucza.key"
$hasloZaszyfrowaneAES = 'HasloDoZaszyfrowania' | ConvertTo-SecureString -AsPlainText -
Force | ConvertFrom-SecureString -Key $kluczAES
```

Parametr `-Key` zamienia mechanizm szyfrowania z DPAPI (przywiązanego do konta i maszyny) na AES z jawnie podanym kluczem — zaszyfrowany w ten sposób ciąg można odszyfrować na innej maszynie i innym kontem, o ile ma się dostęp do tego samego pliku klucza.

AES = Advanced Encryption Standard: powszechny szyfr blokowy, którym szyfrują dane karty DESFire i wiele innych systemów, w naszych przykładach z kluczem 128 bitów · **DPAPI** = Data

Protection API: mechanizm Windowsa szyfrujący dane kluczem konkretnego użytkownika

□□ TYLKO ODCZYT

Odszyfrowanie hasła zaszyfrowanego kluczem AES

```
$kluczAES = Get-Content "SciezkaDoKlucza.key"
$hasloOdszyfrowane = $hasloZaszyfrowaneAES | ConvertTo-SecureString -Key $kluczAES
```

Symetryczne do szyfrowania — ten sam klucz, ten sam parametr `-Key`, ale w kierunku odwrotnym.

□□ **AES** = Advanced Encryption Standard: powszechny szyfr blokowy, którym szyfrują dane karty DESFire i wiele innych systemów, w naszych przykładach z kluczem 128 bitów

□□ ZMIENIA SYSTEM

Zapisanie sekretu we współczesnym magazynie: Microsoft.PowerShell.SecretManagement

```
Install-Module Microsoft.PowerShell.SecretManagement, Microsoft.PowerShell.SecretStore -Force
Register-SecretVault -Name MojSejf -ModuleName Microsoft.PowerShell.SecretStore -Default
Set-Secret -Name "HasloDoAplikacji" -Secret "WartoscHasla"
```

Rejestrujemy lokalny magazyn sekretów (`SecretStore`) i zapisujemy w nim hasło pod nazwaną etykietą — magazyn sam zarządza szyfrowaniem, opcjonalnym hasłem-kluczem do sejfu i integracją z innymi magazynami (Azure Key Vault, KeePass), zamiast zmuszać nas do ręcznego wybierania między DPAPI a własnym kluczem AES.

□□ **DPAPI** = Data Protection API: mechanizm Windowsa szyfrujący dane kluczem konkretnego użytkownika · **AES** = Advanced Encryption Standard: powszechny szyfr blokowy, którym szyfrują dane karty DESFire i wiele innych systemów, w naszych przykładach z kluczem 128 bitów

□□ TYLKO ODCZYT

Odczytanie sekretu z magazynu

```
$haslo = Get-Secret -Name "HasloDoAplikacji" -AsPlainText
```

Odczytujemy zapisany wcześniej sekret — `-AsPlainText` zwraca gotowy tekst, bez `-AsPlainText` polecenie zwraca `SecureString`, spójnie z resztą API PowerShella opisanego wyżej na tej stronie.

> **Uważaj**
>
> - Hasło zaszyfrowane domyślnym mechanizmem DPAPI działa wyłącznie na tym samym koncie i tej samej maszynie, na której zostało zaszyfrowane — przeniesienie pliku z zaszyfrowanym ciągiem na inny serwer albo próba odczytania go innym kontem systemowym kończy się błędem odszyfrowania, nie odczytaniem złego hasła.
> - Klucz AES zapisany w tym samym repozytorium co skrypt, który go używa, likwiduje sens szyfrowania — ktokolwiek ma dostęp do repozytorium, ma dostęp do obu składników potrzebnych do odzyskania hasła jawnego.
> - `Get-Credential` w sesji uruchomionej bez interakcji z użytkownikiem (np. z zadania zaplanowanego bez okna) zawiesza się na czeka na wejście, którego nikt nie dostarczy — dla skryptów nienadzorowanych potrzebne jest jedno z rozwiązań wyżej na tej stronie, nie interaktywne pytanie.

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **DPAPI** = Data Protection API: mechanizm Windowsa szyfrujący dane kluczem konkretnego użytkownika · **AES** = Advanced Encryption Standard: powszechny szyfr blokowy, którym szyfrują dane karty DESFire i wiele innych systemów, w naszych przykładach z kluczem 128 bitów

Z życia wzięte

Z ŻYCIA WZIĘTE

My, zespół IT, przygotowujemy w piątek po godzinach nowy skrypt do harmonogramu zadań na wsus01, uwierzytelniający się kontem usługowym do wysyłki raportów. Hasło szyfrujemy przez `ConvertFrom-SecureString` na własnym laptopie, wklejamy wynik do skryptu i publikujemy go na serwer. W poniedziałek rano zadanie kończy się błędem `Key not valid for use in specified state`, bo DPAPI odszyfrowuje ciąg tylko na tym koncie i tej maszynie, na której powstał, nigdy na koncie SYSTEM na wsus01.

Po co to istnieje

Skrypt administracyjny prędzej czy później potrzebuje się gdzieś uwierzytelnić — do domeny, do API, do bazy danych — a hasło musi skądś wziąć bez człowieka wpisującego je ręcznie za każdym

uruchomieniem, zwłaszcza jeśli skrypt startuje z harmonogramu zadań o trzeciej w nocy. Najprostsze, najgorsze rozwiązanie to wpisanie hasła wprost w treści skryptu jako zwykły tekst — czytelne dla każdego, kto ma dostęp do pliku, widoczne w historii systemu kontroli wersji na zawsze, nawet jeśli hasło później zmienimy. Reszta tej strony to kolejne, coraz lepsze sposoby uniknięcia tego dokładnie problemu.

> **Pojęcie: DPAPI.** Data Protection API — mechanizm systemu Windows szyfrujący dane kluczem wyprowadzonym z poświadczeń logowania konkretnego konta użytkownika na konkretnej maszynie. `ConvertFrom-SecureString` bez podanego `-Key` korzysta właśnie z DPAPI — stąd zaszyfrowany ciąg jest bezużyteczny dla kogokolwiek innego niż to samo konto, na tej samej maszynie, co jest jednocześnie jego siłą i jego najczęściej mylnie rozumianym ograniczeniem.

Analogia

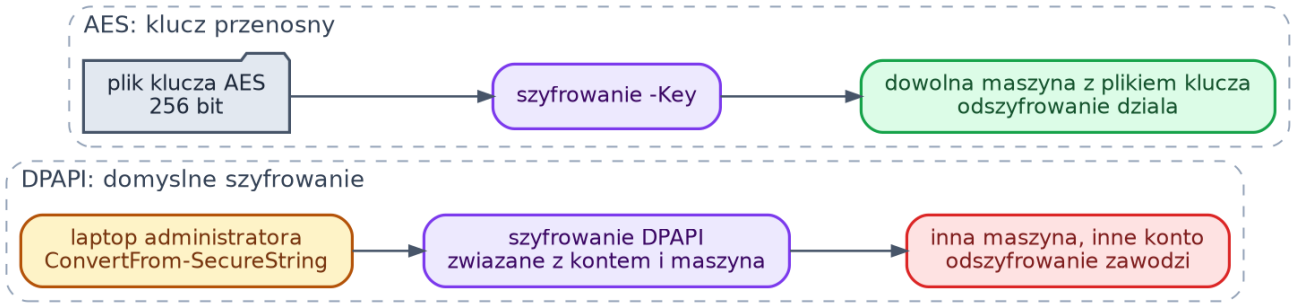
DPAPI działa jak sejf wbudowany w ścianę konkretnego pokoju hotelowego, otwierany odciskiem palca konkretnego gościa. Zawartość jest bezpieczna, dopóki nikt inny nie ma dostępu do tego pokoju i tego odcisku — ale jeśli spróbujemy przenieść ten sam sejf do innego pokoju albo dać klucz innemu gościowi, mechanizm po prostu odmawia otwarcia, bo cały sens jego działania opierał się na tym jednym, konkretnym powiązaniu miejsca i osoby. Klucz AES zapisany osobno to jak zwykła kłódka z kluczem — działa wszędzie, na dowolnym pokoju, ale odpowiedzialność za pilnowanie klucza spada teraz w całości na nas, bo kłódka sama z siebie nie wie, kto powinien mieć do niej dostęp. Ograniczenie tej analogii: przeniesienie sejfu hotelowego do innego pokoju jest fizycznie trudne, podczas gdy skopiowanie pliku z zaszyfrowanym przez DPAPI ciągiem jest trywialne — różnica polega wyłącznie na tym, że skopiowany plik mimo wszystko nie da się otworzyć bez oryginalnego konta i maszyny.

Jak to działa

Rozwiń: jak to działa pod spodem

Najczęstsza pułapka z tej strony ma jedno źródło: administrator generuje zaszyfrowany ciąg, siedząc na własnym koncie, na własnej stacji roboczej, a potem wkleja go do skryptu, który ma uruchamiać się z konta usługi albo z konta systemowego harmonogramu zadań na zupełnie innej maszynie. Odszyfrowanie zawodzi nie dlatego, że hasło jest złe, tylko dlatego, że DPAPI odmawia odszyfrowania czegokolwiek zaszyfrowanego przez inne konto — komunikat błędu (`Key not valid for use in specified state`) rzadko od razu naprowadza na tę przyczynę, bo

brzmi jak problem z samym kluczem, nie z tożsamością wykonawcy. Rozwiązaniem nie jest szukanie „poprawniejszego" sposobu zapisania hasła DPAPI, tylko wygenerowanie go z tego samego konta i tej samej maszyny, która realnie uruchomi skrypt docelowo — albo świadome przejście na klucz AES, jeśli przenośność między maszynami jest wymaganiem.



DPAPI szyfruje na stale dla jednego konta i jednej maszyny, klucz AES daje sie przeniesc na dowolna inna (kliknięcie otwiera pełny rozmiar)

Klucz AES rozwiązuje problem przenośności kosztem przeniesienia całej odpowiedzialności za bezpieczeństwo z systemu operacyjnego na nas — plik klucza staje się jedynym sekretem, którego pilnowanie decyduje o bezpieczeństwie wszystkiego, co nim zaszyfrowano. Uprawnienia systemu plików na pliku klucza (dostęp tylko dla konta uruchamiającego skrypt) są w tym układzie równie krytyczne jak samo istnienie szyfrowania.

`SecretManagement` i `SecretStore` to oficjalny, wspierany przez Microsoft następny krok — moduł `SecretManagement` daje jednolite API (`Get-Secret`, `Set-Secret`) niezależne od tego, gdzie sekret faktycznie leży, a `SecretStore` jest lokalnym magazynem spełniającym tę rolę, z opcjonalnym hasłem chroniącym cały sejf i konfigurowalnym czasem automatycznego blokowania. Ten sam interfejs `SecretManagement` obsługuje też magazyny zewnętrzne (Azure Key Vault, HashiCorp Vault, KeePass) przez osobne rozszerzenia — jeśli organizacja ma już scentralizowany magazyn sekretów, warto sięgnąć po jego rozszerzenie zamiast po lokalny `SecretStore`.

Częste błędy

Wpisanie hasła jawnym tekstem wprost w treści skryptu. Trafia do historii systemu kontroli wersji na zawsze, nawet po późniejszym usunięciu z bieżącej wersji pliku — jedyną skuteczną naprawą po takim wycieku jest zmiana samego hasła, nie usunięcie linijki z kodu.

Wygenerowanie zaszyfrowanego hasła na koncie administratora, uruchomienie skryptu z konta usługi. Odszyfrowanie zawodzi z komunikatem sugerującym problem z kluczem, podczas

gdy realną przyczyną jest niedopasowanie konta — opisane szerzej w sekcji „Jak to działa" wyżej.

Przechowywanie konta usługi z hasłem w pliku konfiguracyjnym zamiast użycia konta zarządzanego. Nawet dobrze zaszyfrowane hasło konta usługi wymaga ręcznej rotacji i niesie ryzyko wycieku pliku z kluczem razem z zaszyfrowanym ciągiem — tam, gdzie infrastruktura na to pozwala, konto usługi zarządzanej (gMSA) eliminuje potrzebę przechowywania hasła w ogóle, bo hasło rotuje automatycznie i nigdy nie jest znane administratorowi ani zapisane w żadnym pliku.

Konfigurację takich kont opisuje strona Konfiguracja kont MSA / gMSA.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell, bez uprawnień administratora.

Ćwiczenie 1 — DPAPI i granica konta

```
$zaszyfrowane = 'HasloTestowe' | ConvertTo-SecureString -AsPlainText -Force | ConvertFrom-SecureString
$zaszyfrowane | Set-Content "$env:TEMP\test_haslo.txt"

# Odczyt tym samym kontem – powinien zadziałać
Get-Content "$env:TEMP\test_haslo.txt" | ConvertTo-SecureString
```

Sprawdzamy, że odszyfrowanie działa na tym samym koncie, na którym powstało zaszyfrowanie.

Odpowiedź

Odczyt się powiedzie i zwróci obiekt `SecureString` bez błędu. Ten sam plik `test_haslo.txt` przesłany na inne konto użytkownika albo na inną maszynę i odczytany identycznym poleceniem `ConvertTo-SecureString` (bez `-Key`) zwróci błąd `Key not valid for use in specified state` — to jest praktyczna demonstracja granicy DPAPI opisanej w tej fiszce.

Ćwiczenie 2 — porównanie DPAPI i AES

```
$klucz = New-Object Byte[] 32
[Security.Cryptography.RNGCryptoServiceProvider]::Create().GetBytes($klucz)
```

```
$zaszyfrowaneDPAPI = 'HasloTestowe' | ConvertTo-SecureString -AsPlainText -Force |
ConvertFrom-SecureString
$zaszyfrowaneAES = 'HasloTestowe' | ConvertTo-SecureString -AsPlainText -Force | ConvertFrom-
SecureString -Key $klucz

$zaszyfrowaneDPAPI.Length
$zaszyfrowaneAES.Length
```

Porównujemy oba podejścia i zastanawiamy się, co potrzeba, żeby odszyfrować każdy z tych dwóch ciągów na innej maszynie.

Odpowiedź

Ciąg zaszyfrowany DPAPI da się odszyfrować wyłącznie na tym samym koncie i tej samej maszynie — nie ma znaczenia, ile razy go skopiujemy, bez oryginalnego konta pozostaje bezużyteczny. Ciąg zaszyfrowany kluczem AES da się odszyfrować wszędzie, pod warunkiem posiadania pliku z kluczem `$klucz` — przenośność zyskujemy kosztem odpowiedzialności za osobne, bezpieczne przechowanie tego klucza, o czym mówi sekcja „Jak to działa".

Pytania kontrolne

1. Administrator zaszyfrował hasło do konta serwisowego na swoim laptopie i wkleił zaszyfrowany ciąg do skryptu uruchamianego z harmonogramu zadań na serwerze, pod kontem gMSA. Dlaczego skrypt nie zadziała, mimo że hasło jest poprawne?

Odpowiedź

Zaszyfrowanie DPAPI wiąże wynikowy ciąg z kontem i maszyną, na której powstał — laptopem administratora, nie kontem gMSA na serwerze. Odszyfrowanie może wykonać wyłącznie to samo konto na tej samej maszynie, więc harmonogram zadań uruchamiający skrypt jako gMSA na serwerze nigdy nie odszyfruje ciągu zaszyfrowanego na laptopie, niezależnie od poprawności samego hasła. Zresztą przy koncie gMSA problem znika całkowicie z innego powodu — konto zarządzane nie wymaga przechowywania żadnego hasła w skrypcie.

2. Dlaczego SecretManagement z SecretStore jest zalecane zamiast ręcznego wyboru między DPAPI a własnym kluczem AES?

Odpowiedź

SecretStore rozwiązuje decyzję między przenośnością a wygodą inaczej niż ręczne DPAPI/AES — daje jednolite, przetestowane API niezależne od tego, jaki mechanizm szyfrowania stoi pod spodem, wspiera opcjonalne hasło do całego sejfu i automatyczne blokowanie po czasie bezczynności, i integruje się z zewnętrznymi magazynami sekretów bez zmiany kodu skryptu korzystającego z `Get-Secret`. Ręczne zarządzanie plikami z zaszyfrowanymi ciągami i osobnymi plikami kluczy wymaga odtworzenia tej samej dyscypliny ręcznie, z większym ryzykiem pomyłki przy każdym nowym skrypcie.

Powiązane: Skrypty PowerShell: struktura i dobre praktyki · WinRM: zarządzanie zdalne · Konfiguracja kont MSA / gMSA

Źródła: `Get-Help about_PSCredential` · `Get-Help ConvertFrom-SecureString` · dokumentacja Microsoft Learn, moduł „Microsoft.PowerShell.SecretManagement"