
# Iron Scripter 2018: Prequel 4

# A commentary

## The Puzzle

Greetings Iron Scripters. When you complete this challenge, you'll be over a quarter of the way to Iron Scripter.

There are many command line tools still in use that don't have PowerShell equivalents. For this challenge you are required to generate a PowerShell equivalent to the output of netstat.exe. The code should output objects that can be used for sorting and filtering. You should then do the same for the arp.exe utility. Again, the expected output is objects that can be used on the PowerShell pipeline.

The functionality should be delivered as a module that can be easily distributed.

Use best practice if it doesn't conflict with your faction's aims. The solution must be acceptable to your faction:

- Daybreak Faction - beautiful code
- Flawless Faction - flawless code
- Battle Faction - good enough to get the job done

As in previous challenges the standard is PowerShell v5.1. Will the code work on PowerShell v6?

Good luck and good coding.

## The Commentary

A short puzzle but the basic idea is to create a module with functions that return the information from legacy utilities – netstat and arp – as objects that can be used for further processing in PowerShell.

Before I explain how to do that I should point out that it turns out there are cmdlets that perform similar functions to netstat and arp. Get-NetTCPConnection supplies similar information to netstat as shown in figure 1.

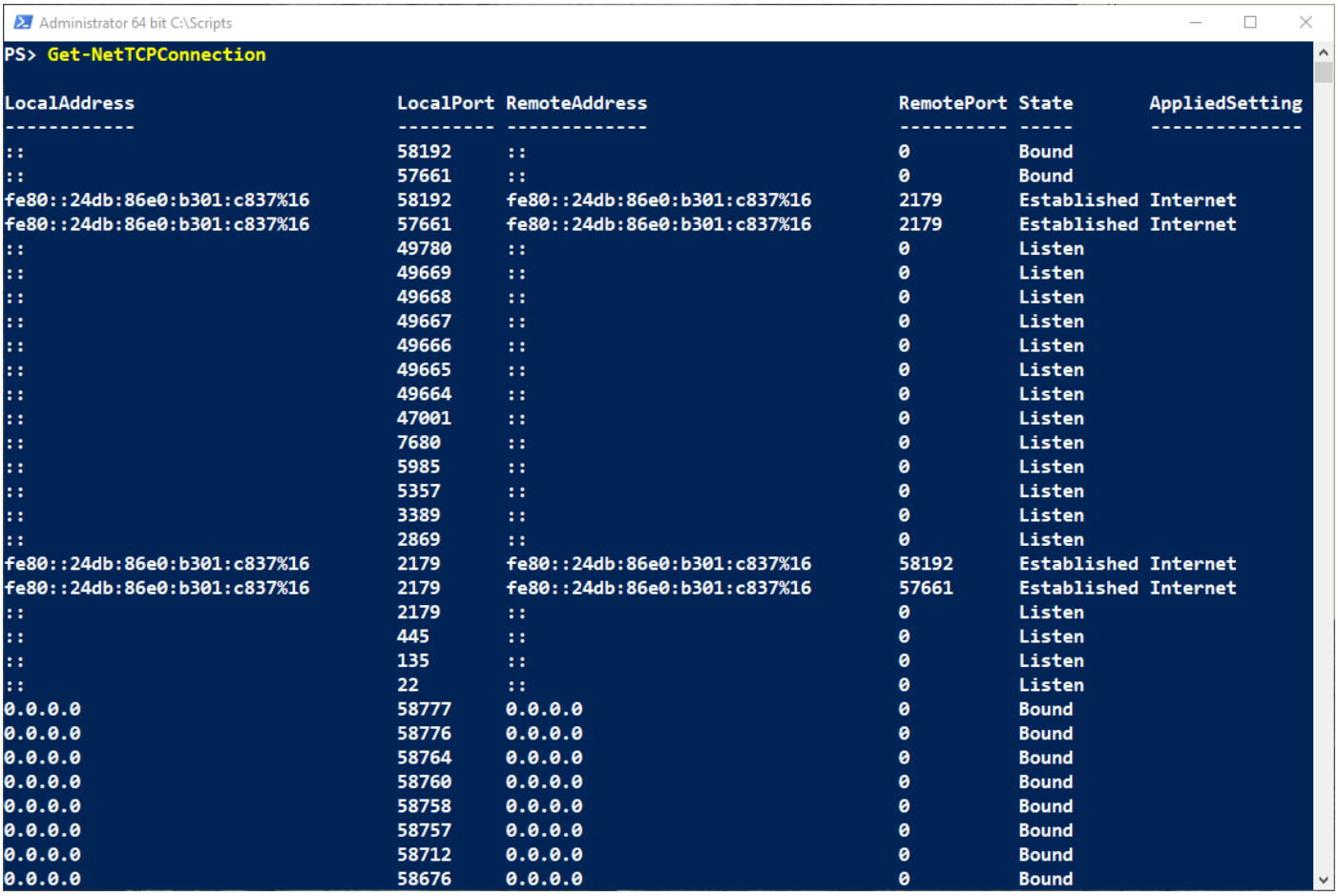


Figure 1 Using Get-NetTCPConnection

The equivalent of arp is Get-NetNeighbor – see figure 2

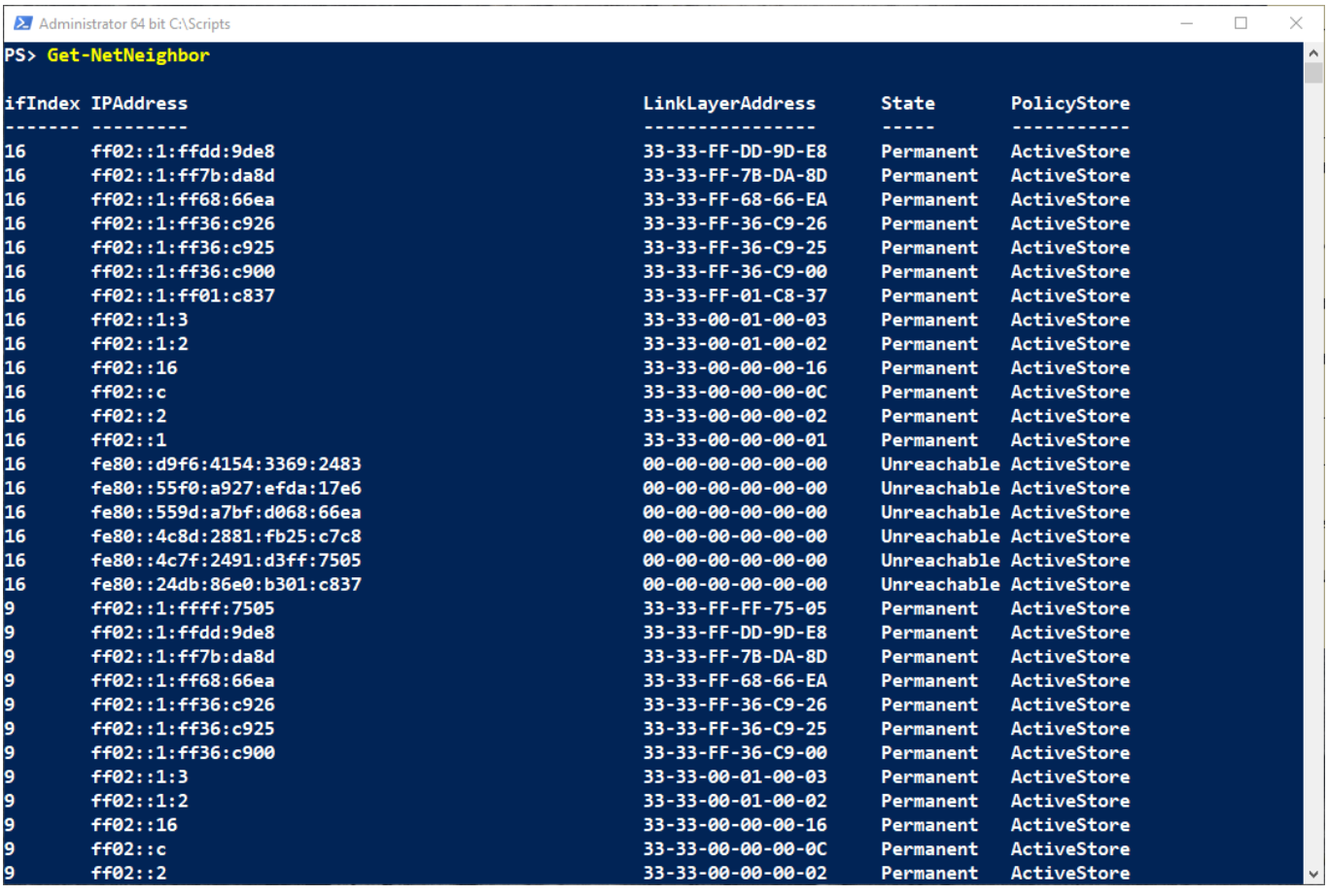


Figure 2 Using Get-NetNeighbor

Both of these cmdlets are in the NetTCPIP module which is a CDXML module introduced with Windows 8 / Windows Server 2012. The cmdlets aren't available on earlier platforms even if you install WMF 3, 4, 5 or 5.1.

By default, these cmdlets aren't available in PowerShell v6. If you add the PowerShell v5.1 modules to your module path:

```
PS>  $env:PSModulePath = 'C:\Windows\System32\WindowsPowerShell\v1.0\Modules\;' + $env:PSModulePath
```

You can access them.

That would solve the spirit of puzzle. I didn't realise Get-Neighbor existed (or that neighbour was incorrectly spelled) when I wrote the puzzle so what I had in mind was using ConvertFrom-String, which was introduced in PowerShell v5 to convert netstat's structured text data into objects.

The output (first few lines only) from netstat looks like this:

```
PS> netstat

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    192.168.0.4:56160      db5sch101101430:https  ESTABLISHED
  TCP    192.168.0.4:56259      db5sch101101722:https  ESTABLISHED
  TCP    192.168.0.4:56270      ec2-35-176-125-18:https  ESTABLISHED
```

The goal is to replace this with an object with the same properties. Netstat can take a little while to run so, for development, I'm going to cheat and capture the output from netstat

```
$stats = netstat
```

Once the function is working then we'll revert to capturing raw netstat data. The first few lines contain the line 'Active Connections' a blank line and the field headers. We need to skip those lines in our processing.

```
PS> $stats | Select-Object -Skip 4

  TCP    192.168.0.4:56160      db5sch101101430:https  ESTABLISHED
  TCP    192.168.0.4:56259      db5sch101101722:https  ESTABLISHED
  TCP    192.168.0.4:56270      ec2-35-176-125-18:https  ESTABLISHED
```

Piping the raw data into ConvertFrom-String gives this

```
PS> $stats |
Select-Object -Skip 4 |
ConvertFrom-String


P1 :
P2 : TCP
P3 : 192.168.0.4:56160
P4 : db5sch101101430:https
P5 : ESTABLISHED
```

By default ConvertFrom-String splits the string data on spaces. This makes P1 a zero-length field. You can use the string method Trim() to remove the leading blanks:

```
$stats |
Select-Object -Skip 4 |
ForEach-Object {
 $_.Trim() | ConvertFrom-String
}
```

When you run this the output is closer to what we want

```
PS> $stats |
Select-Object -Skip 4 |
ForEach-Object {
 $_.Trim() | ConvertFrom-String
}

P1  P2                 P3                      P4
--  --                 --                      --
TCP 192.168.0.4:56160  db5sch101101430:https   ESTABLISHED
TCP 192.168.0.4:56259  db5sch101101722:https   ESTABLISHED
TCP 192.168.0.4:56270  ec2-35-176-125-18:https ESTABLISHED
```

Next step is to set the headers to something meaningful

```
$stats |
Select-Object -Skip 4 |
ForEach-Object {
  $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
}
```

```
PS> $stats |
Select-Object -Skip 4 |
ForEach-Object {
  $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
}

Protocol LocalAddress      ForeignAddress          State
-------- ------------      --------------          -----
TCP      192.168.0.4:56160 db5sch101101430:https   ESTABLISHED
TCP      192.168.0.4:56259 db5sch101101722:https   ESTABLISHED
TCP      192.168.0.4:56270 ec2-35-176-125-18:https ESTABLISHED
TCP      192.168.0.4:56341 40.100.173.2:https      ESTABLISHED
```

But what objects are we getting out of this?

```
PS> $stats |
Select-Object -Skip 4 |
ForEach-Object {
  $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
} | Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name           MemberType   Definition
----           ----------   ----------
Equals         Method       bool Equals(System.Object obj)
GetHashCode    Method       int GetHashCode()
GetType        Method       type GetType()
ToString       Method       string ToString()
ForeignAddress NoteProperty string ForeignAddress=db5sch101101430:https
LocalAddress   NoteProperty string LocalAddress=192.168.0.4:56160
Protocol       NoteProperty string Protocol=TCP
State          NoteProperty string State=ESTABLISHED
```

Let's give the object a more meaningful name

```
$stats |
Select-Object -Skip 4 |
ForEach-Object {
  $stat = $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
  $stat.PSTypeNames[0] = 'Networking.NetStat'
  $stat
}
```

```
PS> $stats |
Select-Object -Skip 4 |
ForEach-Object {
  $stat = $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
  $stat.PSTypeNames[0] = 'Networking.NetStat'
  $stat
```

```
} | Get-Member


   TypeName: Networking.NetStat

Name           MemberType   Definition
----           ----------   ----------
Equals         Method       bool Equals(System.Object obj)
GetHashCode    Method       int GetHashCode()
GetType        Method       type GetType()
ToString       Method       string ToString()
ForeignAddress NoteProperty string ForeignAddress=db5sch101101430:https
LocalAddress   NoteProperty string LocalAddress=192.168.0.4:56160
Protocol       NoteProperty string Protocol=TCP
State          NoteProperty string State=ESTABLISHED
```

Converting this to a function is simple

```
function Get-NetStat {

$stats |
Select-Object -Skip 4 |
ForEach-Object {
  $stat = $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
  $stat.PSTypeNames[0] = 'Networking.NetStat'
  $stat
}

}
```

As you can see from these examples the objects behave on the pipeline.

```
PS> Get-NetStat | select -First 3

Protocol LocalAddress      ForeignAddress          State
-------- ------------      --------------          -----
TCP      192.168.0.4:56160 db5sch101101430:https   ESTABLISHED
TCP      192.168.0.4:56259 db5sch101101722:https   ESTABLISHED
TCP      192.168.0.4:56270 ec2-35-176-125-18:https ESTABLISHED
```

```
PS> Get-NetStat | select -First 3

Protocol LocalAddress      ForeignAddress          State
-------- ------------      --------------          -----
TCP      192.168.0.4:56160 db5sch101101430:https   ESTABLISHED
TCP      192.168.0.4:56259 db5sch101101722:https   ESTABLISHED
TCP      192.168.0.4:56270 ec2-35-176-125-18:https ESTABLISHED
```

```
PS> Get-NetStat | where LocalAddress -like "*:59381"

Protocol LocalAddress      ForeignAddress       State
-------- ------------      --------------       -----
TCP      192.168.0.4:59381 40.101.125.226:https ESTABLISHED
```

Replace $stats with netstat and the first function is done:

```
function Get-NetStat {

netstat |
Select-Object -Skip 4 |
ForEach-Object {
  $stat = $_.Trim() |
  ConvertFrom-String -PropertyNames 'Protocol', 'LocalAddress', 'ForeignAddress', 'State'
  $stat.PSTypeNames[0] = 'Networking.NetStat'
  $stat
}

}
```

Next, we need to perform a similar process with arp. Of course, the output from arp is totally different. One of the joys of the legacy utilities is their different approaches to output. PowerShell really spoils us with its consistency of outputting objects.

```
PS> arp -a

Interface: 172.23.143.161 --- 0x9
  Internet Address      Physical Address      Type
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static

Interface: 169.254.199.200 --- 0xc
  Internet Address      Physical Address      Type
  169.254.255.255       ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

In this case we'll to process the text depending on the data. The first line to be output is blank so should be skipped. Lines like

```
Interface: 172.23.143.161 --- 0x9
```

Need to be processed to extract the IP address and interface ID.

Lines like

```
  Internet Address      Physical Address      Type
```

Need to be skipped

And lines like

```
224.0.0.22            01-00-5e-00-00-16     static
```

Need to be processed as objects.

The code ends up like this:

```powershell
function Get-Arp {
[CmdletBinding()]
$ipaddress = ''
$interface = 0

arp -a |
foreach {
  $arp = $PSItem.Trim()

  if ($arp.StartsWith('Interface:')) {
                                      $intfs = ($_ -split ': ' ) -split '--- '
                                      $interface = [int]$intfs[2]
                                      $ipaddress = $intfs[1]
                                    }

  Write-Verbose $arp
  switch ($arp) {

    {$_ -eq ''}                   { break}

    {$_.StartsWith('Interface:')}  { break  }

    {$_.StartsWith('Internet')}  { break }

    default                     {
                                    $obj = $_ |
                                    ConvertFrom-String -PropertyNames 'InternetAddress', 'PhysicalAddress', 'Type' |
                                    Add-Member -MemberType NoteProperty -Name 'IPAddress' -Value $ipaddress -PassThru |
                                    Add-Member -MemberType NoteProperty -Name 'Interface' -Value $interface -PassThru

                                    $obj.PSTypeNames[0] = 'Networking.Arp'
                                    $obj
                                }
  }
}
}
```

Set the $ipaddress and $interface variables. Run arp and foreach line in the output trim off blank spaces. If the line starts with Interface: split it and populate the $interface and $ipaddress variables.

Pass the trimmed string data into a PowerShell switch statement. If the line is empty, starts with Interface: or Internet then skip it. Otherwise, use ConvertFrom-String to get an object and Add-Member to add the IPAddress and Interface properties. Set the object type and output.

When you run it you get something like this:

```
PS> Get-Arp | Format-Table

InternetAddress PhysicalAddress   Type    IPAddress        Interface
--------------- ---------------   ----    ---------        ---------
224.0.0.22      01-00-5e-00-00-16 static  172.23.143.161           9
224.0.0.252     01-00-5e-00-00-fc static  172.23.143.161           9
239.255.255.250 01-00-5e-7f-ff-fa static  172.23.143.161           9
255.255.255.255 ff-ff-ff-ff-ff-ff static  172.23.143.161           9
169.254.255.255 ff-ff-ff-ff-ff-ff static  169.254.199.200         12
224.0.0.22      01-00-5e-00-00-16 static  169.254.199.200         12
224.0.0.252     01-00-5e-00-00-fc static  169.254.199.200         12
239.255.255.250 01-00-5e-7f-ff-fa static  169.254.199.200         12
255.255.255.255 ff-ff-ff-ff-ff-ff static  169.254.199.200         12
172.22.191.255  ff-ff-ff-ff-ff-ff static  172.22.176.1           16
224.0.0.22      01-00-5e-00-00-16 static  172.22.176.1           16
224.0.0.252     01-00-5e-00-00-fc static  172.22.176.1           16
239.255.255.250 01-00-5e-7f-ff-fa static  172.22.176.1           16
255.255.255.255 ff-ff-ff-ff-ff-ff static  172.22.176.1           16
192.168.0.1     2c-b0-5d-63-77-d6 dynamic 192.168.0.4            21
192.168.0.255   ff-ff-ff-ff-ff-ff static  192.168.0.4            21
224.0.0.22      01-00-5e-00-00-16 static  192.168.0.4            21
224.0.0.252     01-00-5e-00-00-fc static  192.168.0.4            21
239.255.255.250 01-00-5e-7f-ff-fa static  192.168.0.4            21
255.255.255.255 ff-ff-ff-ff-ff-ff static  192.168.0.4            21
```

The objects produced by Get-Arp work on the pipeline.

```
PS> Get-Arp | where Interface -eq 9 | Format-Table

InternetAddress PhysicalAddress   Type   IPAddress       Interface
--------------- ---------------   ----   ---------       ---------
224.0.0.22      01-00-5e-00-00-16 static 172.23.143.161          9
224.0.0.252     01-00-5e-00-00-fc static 172.23.143.161          9
239.255.255.250 01-00-5e-7f-ff-fa static 172.23.143.161          9
255.255.255.255 ff-ff-ff-ff-ff-ff static 172.23.143.161          9
```

Add both functions to a module file - .psm1 - and save with a name of your choice. This code won't work on PowerShell v6 because ConvertFrom-String isn't present. You can use the cmdlets discussed earlier.

As usual, Battle faction will probably stick with what we have. It meets the requirements and gets the job done.

Daybreak faction will want to format the code so that it looks good. Maybe add a module manifest. Possibly add other netstat or arp switches as options.

Flawless faction will want to add all the bells and whistles. All the options from Daybreak faction plus help file, parameter validation, Write-Debug and Write-Verbose commands as appropriate, try-catch blocks as necessary and anything else that ensures the code executes flawlessly.

Enjoy!

Puzzle 5 will be available around the time you're reading this.