Building Netcat with PowerShell

1

Posted by Jason Stangroome on October 3, 2014 | Rate it

| 5,382 views

g+1 16

I have been a long time Windows user and a PowerShell fan since it was introduced. I have always been aware of the range of useful tools regularly available to Unix/Linux users, but recently I started a new job where everything is built on Linux and Windows is barely mentioned. I now get to use all the tools Linux has to offer on a daily basis, but I still miss the rich metadata PowerShell passes along the pipe.

One tool I've found particularly handy in Linux is Netcat. While Netcat is capable of performing a variety of tasks, in essence it takes data from STDIN, forwards it over a TCP connection to a nominated server and port, and writes any response from the server to STDOUT.

Netcat is useful for issuing requests to mail servers, web servers, software or hardware control ports, or almost any network-exposed service. For me however, Netcat is made most useful for two reasons: It is easily used within a script and it is installed-by-default in most Linux distributions I've used.

The tool most similar to Netcat to be included with Windows is the command-line Telnet Client, but it is not easily scriptable and in recent Windows versions it is an optional feature that needs to be intentionally installed.

With PowerShell's tight integration with the full .NET Framework it is easy to quickly implement at least the basic behaviour of Netcat on Windows using the TcpClient and an Encoding. So easy, that I have written such a script, about 70 lines long (including formatting) in a little over an hour.

My script consists of essentially six sections. First is the parameter block, defining three mandatory parameters, and two optional. The three mandatory parameters are the name (or address) of the destination computer, the destination TCP port number, and the data to send. Technically the "Data" parameter is not marked as mandatory, but it would be somewhat pointless to omit it.

```powershell
function Send-NetworkData {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string]
        $Computer,

        [Parameter(Mandatory)]
        [ValidateRange(1, 65535)]
        [Int16]
        $Port,

        [Parameter(ValueFromPipeline)]
        [string[]]
        $Data,

        [System.Text.Encoding]
        $Encoding = [System.Text.Encoding]::ASCII,

        [TimeSpan]
        $Timeout = [System.Threading.Timeout]::InfiniteTimeSpan
    )
```

The two remaining optional parameters are the text encoding method and the response timeout which default to ASCII, and infinite respectively. While it was tempting to just hard-code these items, in PowerShell it's just as easy to expose them as parameters and a user is likely to want to override these.

POPULAR LATEST COMMENTS

Posh-SSH: Open Source SSH PowerShell Module
38 COMMENTS

Find an unused drive letter
34 COMMENTS

Convert a string to a character array
30 COMMENTS

Downloads

Windows Management Framework – PowerShell 2.0 (All Platforms)

Windows Management Framework 3.0

Windows PowerShell 3.0 and Server Manager Quick Reference Guides

Windows Management Framework 4.0

Windows Management Framework 5.0 Preview

Windows PowerShell 4.0 and Other Quick Reference Guides

Post Tags

"PowerShell Security Special" Active Directory Azure Book Brainteaser conference Convert Debug Deep Dive DeepDive DSC eBook editorial exchange Free getting started Hyper-V Idera infosec interview ise kemp Module NET news PowerShell PowerShell 2.0 PowerShell 3.0 PowerShell 4.0

The second section of the script, is the "begin" block. I have separated the function body into the "begin", "process", and "end" blocks because I want to support the input data being piped in, just as someone might pipe the output of one program into Netcat in Linux.

```
begin {
    # establish the connection and a stream writer
    $Client = New-Object -TypeName System.Net.Sockets.TcpClient
    $Client.Connect($Computer, $Port)
    $Stream = $Client.GetStream()
    $Writer = New-Object -Type System.IO.StreamWriter -ArgumentList $Stream, $Encodi
}
```

In the begin block, I create a new TcpClient object, tell it to connect to the specified computer and port, and setup the StreamWriter object to be used for sending the data in the next section. I'm using a new .NET 4.5 constructor overload for StreamWriter so that it won't close the underlying NetworkStream when I close the StreamWriter. I need this so I can still read the response from the stream.

The third section is the "process" block where I actually send the data on the network. Depending on how the user calls my function, the process block will be called in two different ways. If the user pipes data into my function, the process block will be called once for each item in the pipe. However, if the user calls my function and passes the data to the "Data" parameter directly, the process block will be called once only. Using the PowerShell "foreach" statement here handles both scenarios easily.

```
process {
    # send all the input data
    foreach ($Line in $Data) {
        $Writer.WriteLine($Line)
    }
}
```

The fourth section, is the beginning of the "end" block. At this point, all the user-provided data has been received and then written to the network socket. All this section does is flush any buffered data, dispose the StreamWriter object, and shutdown the sending half of the TCP socket so the destination computer knows we're done sending.

```
end {
   # flush and close the connection send
   $Writer.Flush()
   $Writer.Dispose()
   $Client.Client.Shutdown('Send')
```

The fifth, and most complicated, section is responsible for receiving the response data from the server, if any. First, we configure the Stream with the maximum time to wait for a response. Next we create an empty string to hold the ultimate result, and a byte array buffer for reading raw chunks of the response from the stream.

```
# read the response
        $Stream.ReadTimeout = [System.Threading.Timeout]::Infinite
        if ($Timeout -ne [System.Threading.Timeout]::InfiniteTimeSpan) {
            $Stream.ReadTimeout = $Timeout.TotalMilliseconds
        }

        $Result = ''
        $Buffer = New-Object -TypeName System.Byte[] -ArgumentList $Client.ReceiveB
        do {
            try {
                $ByteCount = $Stream.Read($Buffer, 0, $Buffer.Length)
            } catch [System.IO.IOException] {
                $ByteCount = 0
            }
            if ($ByteCount -gt 0) {
                $Result += $Encoding.GetString($Buffer, 0, $ByteCount)
            }
        } while ($Stream.DataAvailable -or $Client.Client.Connected)

        Write-Output $Result
```

Then, within a loop we read as much data as the buffer will hold, or as much data as there is available to read. If we receive some data we use the configured text encoding to convert the raw bytes to text and append it to the result. If an exception is thrown whilst reading data (typically because the read timeout expired) then we simply treat it as though no data was available.

We then check to see if there is still more data waiting to be read from the stream, or if the socket is still connected, and repeat the loop if either of these is true. If not, we write the aggregated result text to the

standard output pipe.

```
            # cleanup
        $Stream.Dispose()
        $Client.Dispose()
    }

}
```

In the sixth, and final section, the end of the "end" block and the end of the script, the Stream and TcpClient are both disposed. And that is it. There are probably many scenarios not well handled by this part of the script, but it copes with the simple situations I needed it for. For example, sending a simple HTTP 1.0 request to a web server and seeing the response:

```
'GET / HTTP/1.0', '' | Send-NetworkData -Computer www.powershellmagazine.com -Port 8
```

There is much more that would be required to re-implement properly. Sending UDP instead of TCP, alternate line-endings, binary data, send delays, broadcasting to an array of ports, and support for SOCKS proxies would cover most of it. Netcat also supports listening on a port for incoming data as an ad-hoc server, but this would be best implemented by a separate PowerShell cmdlet, probably with a name starting with the "Receive" verb.

There is also room to make my script much more PowerShell-idiomatic. At the least, streaming the network response out as it arrives should be a useful (and reasonably easy) exercise for you, the reader. Other improvements may include returning the network response in objects containing timing or other connection metadata in addition to the response data itself.

I suspect others in the PowerShell community have already implemented more of the Netcat functionality in their own PowerShell scripts than I have, and probably with fewer bugs. So for serious scenarios it would be worth looking around.

However, if you simply wanted to see how to put something together quickly in PowerShell for sending network data, hopefully my script has served its purpose.

The full script, with more examples is available as a Gist on GitHub here:
https://gist.github.com/jstangroome/9adaa87a845e5be906c8

Filed in: Articles, Online Only **Tags:** Netcat, PowerShell

About Jason Stangroome

Jason has been a developer on the Microsoft platform for over 15 years, with a regular tendency to cross in to the infrastructure space. This year Jason took a new job to work with the Linux platform full-time instead. His passion for PowerShell however keeps him coming back to Windows in his spare time, albeit with a fresh perspective.

View all posts by Jason Stangroome →

One Pingback/Trackback

06 October 2014 at 8:10am

[…] Building Netcat with PowerShell (Jason Stangroome) …

Dew Drop – October 6, 2014 (#1870) | Morning Dew

One Response to "Building Netcat with PowerShell"

Leave a Reply

Name (Required)

Mail (will not be published) (Required)

Website

Submit Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

Search

Search this site...

Categories

Select Category

Archives

Select Month

Meta

- Log in
- Entries RSS
- Comments RSS
- WordPress.org

Proudly designed by Theme Junkie.