

## Learn Powershell | Achieve More

*What is this Powershell of which you speak?*

---

## Building a TCP Server Using PowerShell

Posted on [February 22, 2014](#)

Something that I have been working on for the past week or so is building a TCP server that I can use to issue commands from remotely and have it carry out on the remote server.

I've already written a [PowerShell Chat server](#) and [chronicled that build](#), but this is something a little different. I will not be handling more than one client connection and I will instead be running commands on the remote system based on what I send to the open port.

Before I begin, I need to lay out some requirements that I want to hit if I trust that this will be usable in something other than a lab environment.

- Provide some sort of authentication mechanism
  - Also want to impersonate remote client token to run commands
- Capable of returning actual objects from remote system (serialization)
- Handle a single connection and then close connection after the command issued and output returned to client

With that in mind, I can now start out by initializing my server. **Note** that I will be jumping back and forth between a **Client** and **Server** with my code examples. I am also using two separate systems to show a more realistic approach as to how this works.

### Serialization of Data

Before I dive into the TCP side of the house, I want to quickly cover what how we are going to receive the output data from the remote system over the network. If you work with PowerShell remoting at all, you know that serialization plays a major role in receiving the data from a remote system.

In PowerShell V3+, we have the [System.Management.Automation.PSSerializer] class publicly available to us and the appropriate Serialize() and Deserialize() methods available to us to transform the data into XML prior to shipping across the network.

```
$data = Get-ChildItem -File |  
Select -First 1 -Property FullName, Length, LastWriteTime  
  
$serialized = [System.Management.Automation.PSSerializer]::Serialize($Data)  
  
$serialized
```

```
PS C:\Users\Administrator> $data = Get-ChildItem -File |
Select -First 1 -Property FullName, Length, LastWriteTime

$serialized = [System.Management.Automation.PSSerializer]::Serialize($Data)

$serialized
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="0">
    <TN RefId="0">
      <T>Selected.System.IO.FileInfo</T>
      <T>System.Management.Automation.PSCustomObject</T>
      <T>System.Object</T>
    </TN>
    <MS>
      <S N="FullName">C:\Users\Administrator\.gitconfig</S>
      <I64 N="Length">50</I64>
      <DT N="LastWriteTime">2013-07-28T22:03:30.2683427-05:00</DT>
    </MS>
  </Obj>
</Objs>
```

```
$deserialized = [System.Management.Automation.PSSerializer]::Deserialize($serialized)
$deserialized
```

```
PS C:\Users\Administrator> $deserialized = [System.Management.Automation.PSSerializer]::Deserialize($serialized)
$deserialized

FullName                               Length LastWriteTime
-----
C:\Users\Administrator\...             50 7/28/2013 10:03:30 PM
```

Of course, there are distinct possibilities that systems are still running PowerShell V2 in which case these are not publicly available. Using Reflection, we can still access the methods to serialize and deserialize the data.

Fortunately, the PowerShell Community has done the work for us and has two functions ([ConvertTo-CliXml](#) (**Original Author** Oisin Grehan <[Twitter](#) | [Blog](#)>; **Current Version** Joel Bennett <[Twitter](#) | [Blog](#)> ) and [ConvertFrom-CliXml](#) (**Original Author** David Sjstrand; **Current Version** Joel Bennett)) readily available to use. The remote server that I am using for my demo only has PowerShell V2 on it, so I will be making use of ConvertTo-CliXml.

## Server

First it is time to initialize the port listener on the server so we can start accepting connections.

First let's check to see if the port is opened.

```
PS C:\Documents and Settings\Administrator> netstat -ano | Select-String 1655
```

Nothing opened, now lets open that port up.

```
##Server
[console]::Title = ("Server: $env:Computername <{0}> on $port" -f `
[net.dns]::GetHostAddresses($env:Computername)[0].IPAddressToString
$port=1655
$endpoint = new-object System.Net.IPEndPoint ([system.net.ipaddress]::any, $port)
$listener = new-object System.Net.Sockets.TcpListener $endpoint
$listener.start()
$client = $listener.AcceptTcpClient()
```

```

PS C:\> $port=1655
PS C:\> [console]::Title = "Server: $env:COMPUTERNAME <{0}> on Port $Port" -f [net.dns]::GetHostAddresses($env:COMPUTERNAME)[1].IPAddressToString
PS C:\> $endpoint = new-object System.Net.IPEndPoint ([system.net.ipaddress]::any, $port)
PS C:\> $listener = new-object System.Net.Sockets.TcpListener $endpoint
PS C:\> $listener.start()
PS C:\> $client = $listener.AcceptTcpClient()

PS C:\Users\Administrator> netstat -ano | Select-String 1655

```

Protocol	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:1655	0.0.0.0:0	LISTENING	3940

This is a blocking method meaning that until something makes a connection, this will prevent me from accessing the console.

Now that we are listening, lets make an initial connection from the client side.

## Client

```

##Client
[console]::Title = ("Server: $env:Computername <{0}>" -f `
[net.dns]::GetHostAddresses($env:Computername)[0].IPAddressToString
$port=1655
$server='Boe-PC'
$client = New-Object System.Net.Sockets.TcpClient $server, $port

```

```

PS C:\Documents and Settings\proxh> [console]::Title = "Client: $env:COMPUTERNAME <{0}>" -f [net.dns]::GetHostAddresses($env:COMPUTERNAME)[0].IPAddressToString
PS C:\Documents and Settings\proxh> $remotePort=1655
PS C:\Documents and Settings\proxh> $server='Boe-PC'
PS C:\Documents and Settings\proxh> $client = New-Object System.Net.Sockets.TcpClient $server, $remotePort
PS C:\Documents and Settings\proxh> $stream = $client.GetStream()
PS C:\Documents and Settings\proxh> $negotiateStream = New-Object net.security.NegotiateStream -ArgumentList $stream

```

Connection has been made. We can verify on the server by seeing if the console has opened up and also by running another netstat.

```

PS C:\Users\Administrator> netstat -ano | Select-String 1655

```

Protocol	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:1655	0.0.0.0:0	LISTENING	3940
TCP	192.168.1.18:1655	192.168.1.20:1094	ESTABLISHED	3940

So what happens now? Here is where we make the decision to use a [NegotiateStream](#) vs. a regular network [stream](#). By using a NegotiateStream, we will be able to then provide an Authentication Stream that will be used to transfer client and server authentication data between the two as well as being able to sign and encrypt the data transmission. By using a network stream, anonymous users could easily connect to the remote system and issue commands as the user that started the listener! Not exactly what you want to deal with.

```

$stream = $client.GetStream()
$NegotiateStream = New-Object net.security.NegotiateStream -ArgumentList $stream

```

First I get the network stream by calling the GetStream() method and then use that in the construction of the

NegotiateStream object.

Before I do anything else, I need to kick off the same stream on the server.

## Server

```
$stream = $client.GetStream()
$NegotiateStream = New-Object net.security.NegotiateStream -ArgumentList $stream
#Validate Alternate credentials
Try {
    $NegotiateStream.AuthenticateAsServer(
        [System.Net.CredentialCache]::DefaultNetworkCredentials,
        [System.Net.Security.ProtectionLevel]::EncryptAndSign,
        [System.Security.Principal.TokenImpersonationLevel]::Impersonation
    )
    Write-host "$($client.client.RemoteEndPoint.Address) authenticated as $($NegotiateStream.RemoteIdentity.Name) via $($NegotiateStream.RemoteIdentity.AuthenticationType)" -ForegroundColor Green -Background Black
} Catch {
    Write-Warning $_.Exception.Message
}
```

Same as with the client, I have to construct the NegotiateStream object using the network stream. After that it is time to accept an authentication request from the client.

There are 4 possible parameter sets with the [AuthenticateAsServer\(\)](#) method. [In this instance](#), I am choosing to supply a set of default credentials, making sure that I encrypt and sign the data being transferred and declaring that I will only accept Impersonation as my token impersonation level. Attempts at negotiating anything else will end up with a denied connection.

```
PS C:\> Try {
>> $NegotiateStream.AuthenticateAsServer(
>>     [System.Net.CredentialCache]::DefaultNetworkCredentials,
>>     [System.Net.Security.ProtectionLevel]::EncryptAndSign,
>>     [System.Security.Principal.TokenImpersonationLevel]::Impersonation
>> )
>> Write-host "$($client.client.RemoteEndPoint.Address) authenticated as $($NegotiateStream.RemoteIdentity.Name) via $($NegotiateStream.RemoteIdentity.AuthenticationType)" -ForegroundColor Green -Background Black
>> } Catch {
>>     Write-Warning $_.Exception.Message
>> }
>>
```

Once I call this method, it becomes a blocking call until I attempt authentication from my client.

## Client

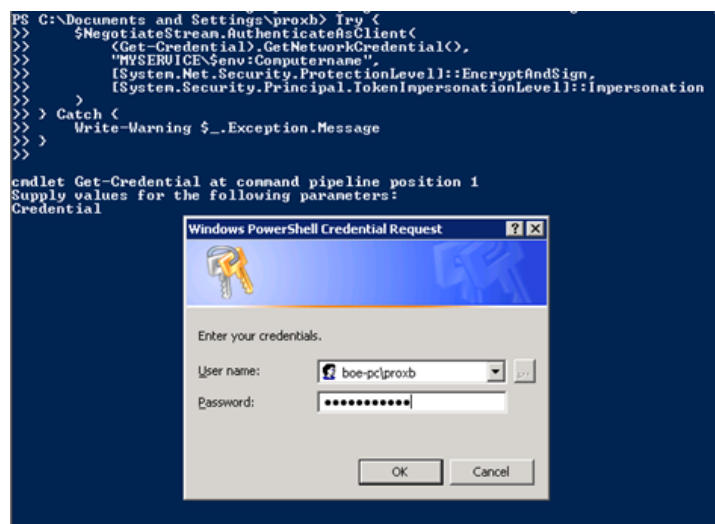
I am now going to use AuthenticateAsClient() to try to pass some alternate credentials (*proxb*) to the server in hopes of being let in.

```
Try {
    $NegotiateStream.AuthenticateAsClient(
        (Get-Credential).GetNetworkCredential(),
        'MYSERVICE\boe-pc',
        [System.Net.Security.ProtectionLevel]::EncryptAndSign,
        [System.Security.Principal.TokenImpersonationLevel]::Impersonation
    )
}
```

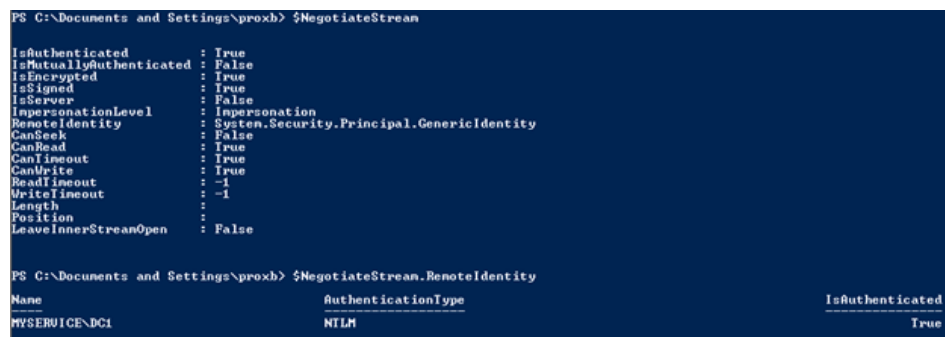
```

} Catch {
    Write-Warning $_.Exception.Message
}

```



The `AuthenticateAsClient()` method is similar to what we used with the Server. It has multiple parameter sets including one where you supply no parameters. Because I want to make sure to negotiate Impersonation with the server, I am making sure to supply my default network credential, a SPN that I made up, `EncryptAndSign` so the data is transmitted securely and finally my `TokenImpersonationLevel` as `Impersonate`.



Inspecting the `NegotiateStream` object, you can see that `IsEncrypted` and `IsSigned` is `True` which is what we wanted as well as the `ImpersonationLevel` is set to `Impersonate`. Diving deeper into the object via the `RemoteIdentity` property, we can see that we are using the SPN that we created as well as the `AuthenticationType`, in this case `NTLM`.

## Server

On the server side, we will also inspect both the `NegotiateStream` and the `RemoteIdentity` property as well to see what they look like.

```

192.168.1.20 authenticated as BOE-PC\proxb via NTLM
PS C:\> $NegotiateStream

IsAuthenticated      : True
IsMutuallyAuthenticated : False
IsEncrypted          : True
IsSigned             : True
IsServer             : True
ImpersonationLevel    : Impersonation
RemoteIdentity        : System.Security.Principal.WindowsIdentity
CanSeek              : False
CanRead              : True
CanTimeout           : True
CanWrite             : True
ReadTimeout          : -1
WriteTimeout         : -1
Length               : 
Position             : 
LeaveInnerStreamOpen  : False

PS C:\> $NegotiateStream.RemoteIdentity

AuthenticationType : NTLM
ImpersonationLevel : Impersonation
IsAuthenticated    : True
IsGuest            : False
IsSystem           : False
IsAnonymous        : False
Name               : BOE-PC\proxb
Owner              : S-1-5-21-4119858167-1669307-1669307-1669307
User               : S-1-5-21-4119858167-1669307-1669307-1669307
Groups             : {S-1-5-21-4119858167-1669307-1669307-1669307
                    S-1-5-2-...}
Token              : 1348
UserClaims         : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name:
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
                    S-1-5-21-4119858167-1669307-1669307-1669307
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
                    S-1-5-21-4119858167-1669307-1669307-1669307
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
                    S-1-5-21-4119858167-1669307-1669307-1669307}
DeviceClaims       : {}
Claims             : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name:
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
                    S-1-5-21-4119858167-1669307-1669307-1669307
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
                    S-1-5-21-4119858167-1669307-1669307-1669307
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
                    S-1-5-21-4119858167-1669307-1669307-1669307}
Actor              : 
BootstrapContext    : 
Label              : 
NameClaimType       : http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
RoleClaimType       : http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsi
  
```

You can see by the Write-Host that I included that the remote client successfully connected as Boe-PC\proxb and that the authentication type was NTLM.

Same information on the NegotiateStream as what was one the client. But the RemoteIdentity property is much different. Here we can see the remote user's name, the token being used as well as the group memberships (shown as a SID) that this user belongs to in relation to this server.

At this point, I could check to see if the remote user has access and make a decision to allow the connection or not. Something like this could be done:

```

([Security.Principal.WindowsPrincipal]$NegotiateStream.RemoteIdentity).IsInRole('Administrators')
  
```

```

PS C:\Documents and Settings\Administrator> ([Security.Principal.WindowsPrincipal]$NegotiateStream.RemoteIdentity).IsInRole('Administrators')
True
  
```

From here I could make a decision based on whether it comes back as True or False to continue allowing the connection or halt it. In this case, I would allow it to continue on.

So now that we have a good connection along with proper authentication, we can continue on with the connection and begin with attempting to impersonate the remote client.

Lets check out who the current user is on the server via the TCP server.

```
[System.Security.Principal.WindowsIdentity]::GetCurrent()
```

```
PS C:\> #ValidateBefore
PS C:\> [System.Security.Principal.WindowsIdentity]::GetCurrent()

AuthenticationType : NTLM
ImpersonationLevel : None
IsAuthenticated    : True
IsGuest            : False
IsSystem           : False
IsAnonymous        : False
Name               : BOE-PC\Administrator
Owner              : S-1-5-32-544
User               : S-1-5-21-4119858167-1669307955-3520882328-500
Groups             : {S-1-5-21-4119858167-1669307955-3520882328-513, S-1-1-0, S-1-5-114,
                    S-1-5-21-4119858167-1669307955-3520882328-1011...}
Token              : 1704
UserClaims         : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name:
                    BOE-PC\Administrator,
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid:
                    S-1-5-21-4119858167-1669307955-3520882328-500,
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid:
                    S-1-5-21-4119858167-1669307955-3520882328-513,
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid:
                    S-1-5-21-4119858167-1669307955-3520882328-513...}
DeviceClaims       : {}
Claims             : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name:
                    BOE-PC\Administrator,
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid:
                    S-1-5-21-4119858167-1669307955-3520882328-500,
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid:
                    S-1-5-21-4119858167-1669307955-3520882328-513,
                    http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid:
                    S-1-5-21-4119858167-1669307955-3520882328-513...}
Actor              :
BootstrapContext   :
Label              :
NameClaimType      : http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
RoleClaimType      : http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid
```

As you can see, I am currently running as my Administrator account (smart, right?) that launched the server. Now I will attempt to impersonate the remote client (*proxb*) and see what happens.

```
$remoteUserToken = $NegotiateStream.RemoteIdentity.Impersonate()
```

```
PS C:\> #Attempt impersonate remote client
PS C:\> $remoteUserToken = $NegotiateStream.RemoteIdentity.Impersonate()
PS C:\> $remoteUserToken
System.Security.Principal.WindowsImpersonationContext
```

I saved the output object (*System.Security.Principal.WindowsImpersonationContext*) as a variable because this will be invaluable later on when I need to stop impersonating the remote client.

```
[System.Security.Principal.WindowsIdentity]::GetCurrent()
```



```

PS C:\> #ValidateAfter
PS C:\> [System.Security.Principal.WindowsIdentity]::GetCurrent()

AuthenticationType : NTLM
ImpersonationLevel : Impersonation
IsAuthenticated : True
IsGuest : False
IsSystem : False
IsAnonymous : False
Name : BOE-PC\proxb
Owner : S-1-5-21-4119858167-1669307955-3520882328-1000
User : S-1-5-21-4119858167-1669307955-3520882328-1000
Groups : {S-1-5-21-4119858167-1669307955-3520882328-513, S-1-1-0, S-1-5-32-545,
S-1-5-2...}
Token : 1492
UserClaims : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: BOE-PC\proxb,
http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid:
S-1-5-21-4119858167-1669307955-3520882328-1000,
http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupid:
S-1-5-21-4119858167-1669307955-3520882328-513,
http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid:
S-1-5-21-4119858167-1669307955-3520882328-513...}
DeviceClaims : {}
Claims : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: BOE-PC\proxb,
http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid:
S-1-5-21-4119858167-1669307955-3520882328-1000,
http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupid:
S-1-5-21-4119858167-1669307955-3520882328-513,
http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid:
S-1-5-21-4119858167-1669307955-3520882328-513...}
Actor :
BootstrapContext :
Label :
NameClaimType : http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
RoleClaimType : http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid

```

As you can see, I am now running under the *boe-pc\proxb* account after a successful impersonation! From here, depending on the rights that you have on the system, you can run commands as the user that you impersonated. Of course, if you have little to no rights, it will be quite difficult to do much of anything.

## Client

Now that we have accomplished the connection and impersonation on the server, I will issue a command that will be run on the remote system and return the results back to the client.

```

$data = [text.Encoding]::Ascii.GetBytes('Get-WMIObject Win32_OperatingSystem | Select __Server, Caption')
Write-Verbose "Sending $($Data.count) bytes"
$NegotiateStream.Write($data,0,$data.length)
$NegotiateStream.Flush()

```



```

PS C:\Documents and Settings\proxb> $data = [text.Encoding]::Ascii.GetBytes('Get-WMIObject Win32_OperatingSystem | Select
__Server, Caption')
PS C:\Documents and Settings\proxb> Write-Verbose "Sending $($Data.count) bytes"
VERBOSE: Sending 62 bytes
PS C:\Documents and Settings\proxb> $NegotiateStream.Write($data,0,$data.length)
PS C:\Documents and Settings\proxb> $NegotiateStream.Flush()
PS C:\Documents and Settings\proxb> _

```

All that I've done is taken the command as a string and converted into bytes using the `GetBytes()` method before sending across the network via the `NegotiateStream` over to the remote system.

## Server

First I will check to see if any data is available:

```
$Stream.DataAvailable
```

```

PS C:\> $Stream.DataAvailable
True

```



I know that there is data available so now I need to begin work to get the data and convert it back into something usable.

```
$StringBuilder = New-Object Text.StringBuilder
Do {
    [byte[]]$byte = New-Object byte[] 1024
    Write-Verbose ("{0} Bytes Left" -f $client.Available)
    $bytesReceived = $NegotiateStream.Read($byte, 0, $byte.Length)
    If ($bytesReceived -gt 0) {
        Write-Verbose ("{0} Bytes received" -f $bytesReceived)
        [void]$StringBuilder.Append([text.Encoding]::Ascii.GetString($byte[0..($bytesReceived - 1)
    } Else {
        $activeConnection = $False
        Break
    }
} While ($Stream.DataAvailable)
```



```
PS C:\> $StringBuilder = New-Object Text.StringBuilder
PS C:\> Do {
>> [byte[]]$byte = New-Object byte[] 1024
>> Write-Verbose ("{0} Bytes Left" -f $client.Available)
>> $bytesReceived = $NegotiateStream.Read($byte, 0, $byte.Length)
>> If ($bytesReceived -gt 0) {
>>     Write-Verbose ("{0} Bytes received" -f $bytesReceived)
>>     [void]$StringBuilder.Append([text.Encoding]::Ascii.GetString($byte[0..($bytesReceived
- 1)))
>> } Else {
>>     $activeConnection = $False
>>     Break
>> }
>> } While ($Stream.DataAvailable)
VERBOSE: 82 Bytes Left
VERBOSE: 62 Bytes received
```

I use the StringBuilder class to handle the reconstruction of the command and continue with a Do loop until all of the data has been received from the remote client. You might be asking why 82 bytes were sent and only 62 bytes are shown to have been received. This is due to the encryption that is being applied prior to sending the data across the network.

With the StringBuilder object, we have to cast it out to a string to see the data inside of it.

```
$StringBuilder.ToString()
```

```
PS C:\> $StringBuilder.ToString()
Get-WMIObject Win32_OperatingSystem | Select __Server, Caption
```

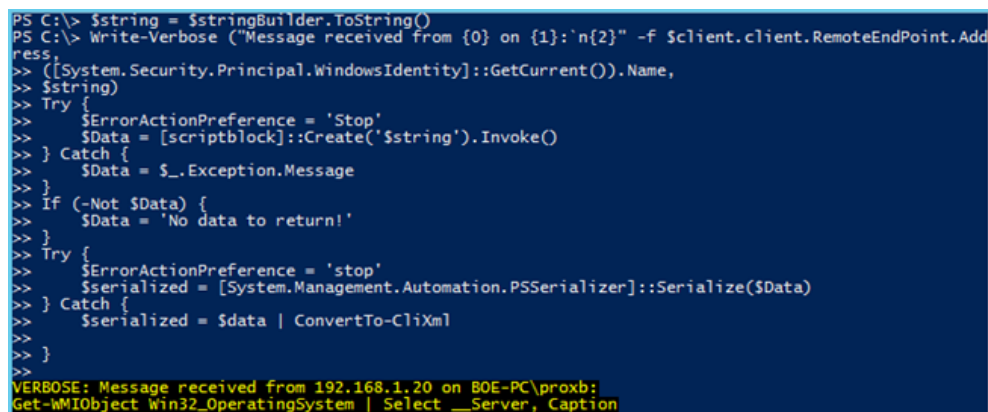
There is our command that was sent from the remote client all ready for us to run here on the remote system. Speaking of which, lets continue on and run this command and save the results.

```
$string = $StringBuilder.ToString()
Write-Verbose ("Message received from {0} on {1}:`n{2}" -f $client.client.RemoteEndPoint.Address,
([System.Security.Principal.WindowsIdentity]::GetCurrent()).Name,
$string)
Try {
    $ErrorActionPreference = 'Stop'
    $Data = [scriptblock]::Create($string).Invoke()
} Catch {
```

```

    $Data = $_.Exception.Message
}
If (-Not $Data) {
    $Data = 'No data to return!'
}
Try {
    $ErrorActionPreference = 'stop'
    $serialized = [System.Management.Automation.PSSerializer]::Serialize($Data)
} Catch {
    $serialized = $data | ConvertTo-CLiXml
}

```



```

PS C:\> $string = $StringBuilder.ToString()
PS C:\> Write-Verbose ("Message received from {0} on {1}:`n{2}" -f $client.client.RemoteEndPoint.Address,
>> ([System.Security.Principal.WindowsIdentity]::GetCurrent()).Name,
>> $string)
>> Try {
>>     $ErrorActionPreference = 'Stop'
>>     $Data = [scriptblock]::Create('$string').Invoke()
>> } Catch {
>>     $Data = $_.Exception.Message
>> }
>> If (-Not $Data) {
>>     $Data = 'No data to return!'
>> }
>> Try {
>>     $ErrorActionPreference = 'stop'
>>     $serialized = [System.Management.Automation.PSSerializer]::Serialize($Data)
>> } Catch {
>>     $serialized = $data | ConvertTo-CLiXml
>> }
>> }
VERBOSE: Message received from 192.168.1.20 on BOE-PC/proxib:
Get-WMIObject Win32_OperatingSystem | Select __Server, Caption

```

I use `[scriptblock]::Create()` to build out the command and then use `Invoke()` to run the command. It's really not much better than using `Invoke-Expression`. With either of these, you need to be mindful of possible code injections.

From there, I need to serialize the object that is being returned so I can send it over to the remote client. First I try the `Serialize()` method and if that fails (I temporarily set the `$erroractionpreference` to `Stop` to make sure the error is terminating), then default to the `ConvertTo-CLiXml` function. The resulting data prior to being sent looks like this:



```

PS C:\> $serialized
<Obj Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04"><Obj RefId="0"><TN RefId="0"><T>Selecte
d.System.Management.ManagementObject</T><T>System.Management.Automation.PSCustomObject</T><T>System.Object</T></TN><MS>
<S N="__SERVER">BOE-PC</S><S N="Caption">Microsoft Windows Server 2012 R2 Datacenter</S></MS></Obj></Obj>

```

Next up is to convert that data into bytes and send it back to the client.

```

$ErrorActionPreference = 'Continue'
#Resend the Data back to the client
$bytes = [text.Encoding]::Ascii.GetBytes($serialized)

#Send the data back to the client
Write-Verbose ("Sending {0} bytes" -f $bytes.count) -Verbose
$NegotiateStream.Write($bytes, 0, $bytes.length)
$NegotiateStream.Flush()

```

```
PS C:\> #Resend the Data back to the client
PS C:\> $bytes = [text.Encoding]::Ascii.GetBytes($serialized)
PS C:\>
PS C:\> #Send the data back to the client
PS C:\> Write-Verbose ("Sending {0} bytes" -f $bytes.count) -Verbose
VERBOSE: Sending 344 bytes
PS C:\> $NegotiateStream.Write($bytes,0,$bytes.length)
PS C:\> $NegotiateStream.Flush()
```

## Client

I am basically going to repeat the same process on the client that I did on the server in checking for any data and then pulling it down as bytes and converting it to a string.

```
$StringBuilder = New-Object Text.StringBuilder
While ($client.available -gt 0) {
    Write-Verbose "Processing Bytes: $($client.Available)" -Verbose
    #$clientstream = $TcpClient.GetStream()
    [byte[]]$inStream = New-Object byte[] $client.Available
    $buffSize = $client.Available
    $return = $NegotiateStream.Read($inStream, 0, $buffSize)
    [void]$StringBuilder.Append([System.Text.Encoding]::ASCII.GetString($inStream[0..($return-1)]))
}
```

```
PS C:\> $StringBuilder = New-Object Text.StringBuilder
PS C:\> While ($Client.Available -gt 0) {
>> Write-Verbose "Processing Bytes: $($Client.Available)" -Verbose
>> $InputStream = $TcpClient.GetStream()
>> [byte[]]$InStream = New-Object byte[] $Client.Available
>> $buffSize = $Client.Available
>> $return = $NegotiateStream.Read($InStream, 0, $buffSize)
>> [void]$StringBuilder.Append([System.Text.Encoding]::ASCII.GetString($InStream[0..($return-1)]))
>> }
VERBOSE: Processing Bytes: 344
```

Let's verify that the data came across.

```
Client: DC1 <192.168.1.20>
PS C:\> $stringBuilder.ToString()
[ObjId Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04"]<?xml:namespace><Obj RefId="0"><TN RefId="0"><T>Select-Object</T><System.Management.Automation.Object</T></T><System.Management.Automation.Object</T></T><System.Object</T></T></T><MS>
<S N="SERVER">BOE-PC</S><S N="Caption">Microsoft Windows Server 2012 R2 Datacenter</S></MS></Obj></Obj></Obj>
```

Now we need to deserialize the data so it is an actual object (PowerShell loves objects! 😊).

```
Try {
    $ErrorActionPreference = 'stop'
    $deserialized = [System.Management.Automation.PSSerializer]::DeSerialize($stringbuilder.ToString())
} Catch {
    $deserialized = $stringbuilder.ToString() | ConvertFrom-CliXml
}
$ErrorActionPreference = 'Continue'
```

Now we can look at the finished product.

```

PS C:\Documents and Settings\prox> Try <
>> $ErrorActionPreference = 'stop'
>> $deserialized = (System.Management.Automation.PSSerializer)::Deserialize($stringbuilder.ToString())
>> > Catch <
>> $deserialized = $stringbuilder.ToString() ! ConvertFrom-Xml
>> >
>> $ErrorActionPreference = 'Continue'
>>
PS C:\Documents and Settings\prox> $deserialized

```

SERVER	Caption
BOE-PC	Microsoft Windows Server 2012 R2 Datacenter

Bear in mind that this is a deserialized object, so you will only have properties and a few select methods such as ToString() available.

```

PS C:\Documents and Settings\prox> $deserialized ! Get-Member

```

TypeName: Deserialized.Selected.System.Management.ManagementObject		
Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
Caption	NoteProperty	System.String Caption=Microsoft Windows Server 2012 R2 Datacenter
__SERVER	NoteProperty	System.String __SERVER=BOE-PC

Now that I am done with my connection, I need to clean up after myself.

## Client

```

$stream.Close()
$client.Close()
$NegotiateStream.Close()
$stream.Dispose()
$client.Dispose()
$NegotiateStream.Dispose()

```

## Server

```

$listener.Stop()
$reader.Dispose()
$stream.Dispose()
$client.Dispose()
$NegotiateStream.Dispose()
$remoteUserToken.Undo()
$remoteUserToken.Dispose()

```

## A More Functional Way to Do This

Working with either of these (client or server) requires a lot of monitoring and manual commands to make sure that you are tracking what is being sent and received between these two connections. Fortunately, I have written a couple of functions that will make this much easier to manage.

First we need to load the module up.

```
Import-Module .\TCPServer.psm1 -Verbose
```

```
PS C:\Users\Administrator\Desktop> ipmo .\TCPServer.psm1 -Verbose
VERBOSE: Loading module from path 'C:\Users\Administrator\Desktop\TCPServer.psm1'.
VERBOSE: Importing function 'Invoke-TCPServer'.
VERBOSE: Importing function 'Send-Command'.
VERBOSE: Importing alias 'itcps'.
VERBOSE: Importing alias 'scmd'.
```

## Invoke-TCPServer

The first function is called Invoke-TCPServer which can be used to either start a TCP server on a local or remote system. You can specify a Computername, Port and a Credential to run the TCP server as (also useful with remote systems that you are starting the server on).

This will start the TCP server and handle one connection at a time as well as using the negotiate stream that I have discussed here. After each connection and command ran based on the client, it will drop the client connection and force another reconnect.

An object is returned when you use the function showing the Computername, Port, ProcessID of the local/remote process being used for the server as well as a port check attempt (an initial warning will display on the server window because it will try to authenticate against the port check; this can be ignored) that you can then reference to track the TCP server.

```
Invoke-TCPServer -Computername 'boe-pc' -Port 1655 `
-Credential 'boe-pc\proxb' -Verbose
```

```
PS D:\SharedStorage> Invoke-TCPServer -Computername 'boe-pc' -Port 1655 -Credential 'boe-pc\administrator' -Verbose
VERBOSE: Assigning values to code block
VERBOSE: Attempting to start TCP Server on boe-pc

Name                Value
----                -
Port                1655
ProcessID           1788
ProcessCreationState Success
Computername        boe-pc
IsPortAvailable     False
```

The IsPortAvailable is kind of hit or miss (I may pull this property in the next release), but the TCP server is now up and running on the remote system. I can verify on the remote system just to be sure.

```
PS C:\> $env:COMPUTERNAME
BOE-PC
PS C:\> netstat -ano | Select-String 1655

TCP 0.0.0.0:1655 0.0.0.0:0 LISTENING 1788
```

For the sake of demonstrating and showing the window, I am going to run the server locally so the window is visible.

```
TCP Server <BOE-PC -- 1655>
VERBOSE: Server started on port 1655
VERBOSE: New connection from 169.254.236.191
VERBOSE: Waiting to authenticate client
WARNING: Exception calling "AuthenticateAsServer" with "3" argument(s): "Unable to read data from the transport connection: The connection was closed."
```

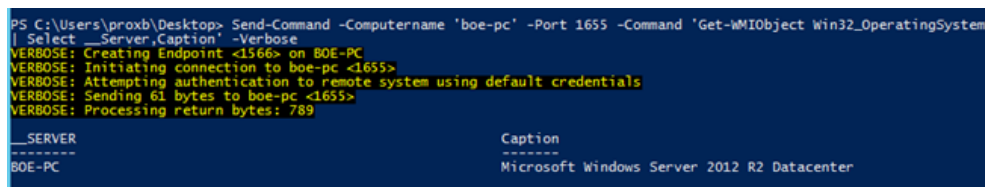
Again, the Warning is more of a friendly warning due to the port check that occurs after starting the process.

## Send-Command

This now leads into my second function, called Send-Command. This does exactly what the function says, sends a command to the remote system and then waits for a response and displays the response (usually an object) to the client console.

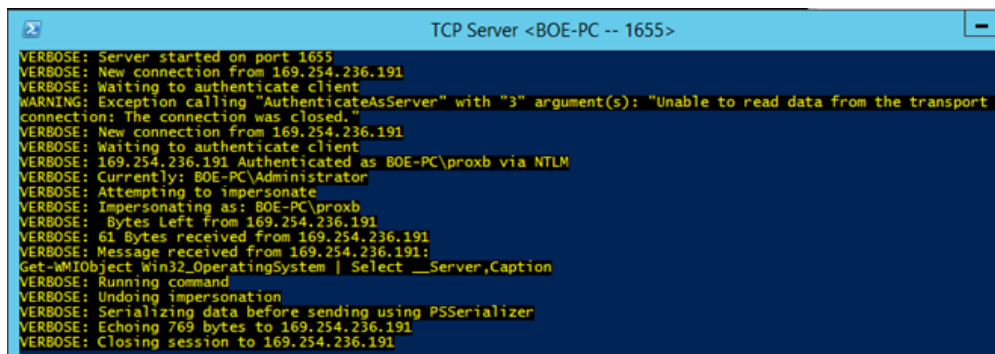
Using my currently running server, lets send a simple command to it and see what happens!

```
Send-Command -Computersname 'boe-pc' -Port 1655 `
-Command 'Get-WMIObject Win32_OperatingSystem | Select __Server, Caption' `
-Verbose -Credential 'boe-pc\proxb'
```



```
PS C:\Users\proxb\Desktop> Send-Command -Computersname 'boe-pc' -Port 1655 -Command 'Get-WMIObject Win32_OperatingSystem
| Select __Server,Caption' -Verbose
VERBOSE: Creating Endpoint <1566> on BOE-PC
VERBOSE: Initiating connection to boe-pc <1655>
VERBOSE: Attempting authentication to remote system using default credentials
VERBOSE: Sending 61 bytes to boe-pc <1655>
VERBOSE: Processing return bytes: 789
__SERVER                                     Caption
-----
BOE-PC                                     Microsoft Windows Server 2012 R2 Datacenter
```

Works rather well. On the client piece, we can see that it attempts authentication and then sends its data to the remote server and waits for a response. Once the response has been received, it presents the data on the console. Now let's look at the server side.



```
TCP Server <BOE-PC -- 1655>
VERBOSE: Server started on port 1655
VERBOSE: New connection from 169.254.236.191
VERBOSE: Waiting to authenticate client
WARNING: Exception calling "AuthenticateAsServer" with "3" argument(s): "Unable to read data from the transport
connection: The connection was closed."
VERBOSE: New connection from 169.254.236.191
VERBOSE: Waiting to authenticate client
VERBOSE: 169.254.236.191 Authenticated as BOE-PC\proxb via NTLM
VERBOSE: Currently: BOE-PC\Administrator
VERBOSE: Attempting to impersonate
VERBOSE: Impersonating as: BOE-PC\proxb
VERBOSE: Bytes Left from 169.254.236.191
VERBOSE: 61 Bytes received from 169.254.236.191
VERBOSE: Message received from 169.254.236.191:
Get-WMIObject Win32_OperatingSystem | Select __Server,Caption
VERBOSE: Running command
VERBOSE: Undoing impersonation
VERBOSE: Serializing data before sending using PSSerializer
VERBOSE: Echoing 769 bytes to 169.254.236.191
VERBOSE: Closing session to 169.254.236.191
```

As you can see, the server took the command, ran it and sent the returned object back to the client. After sending the data back to the client, the connection to the client is closed.

The download link for these two functions are below. Note that these are still Proof of Concept and while they seem like a secure method, it should still be used with caution, especially if used in production. I have plans for more things to add to this, but for now, this is what it is. Feel free to let me know of any bugs or things that you would like to see added.

## Download TCP Server Module

[Technet Script Repository](#)



---

**Share this:**

Be the first to like this.

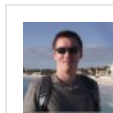
---

**Related**

[Introduction to PoshChat, A PowerShell Chat Client/Server](#)  
In "Modules"

[Fix Duplicate WSUS ClientIDs Using PowerShell](#)  
In "powershell"

[Testing TCP Ports with a Possible Header Response](#)  
In "powershell"

**About Boe Prox**

Microsoft PowerShell MVP working as a Senior Systems Administrator

[View all posts by Boe Prox →](#)

This entry was posted in [powershell](#) and tagged [Powershell](#), [streams](#), [tcp](#). Bookmark the [permalink](#).

## 4 Responses to *Building a TCP Server Using PowerShell*

**dalexandrescu** *says:*

September 9, 2014 at 7:42 am

Cool stuff indeed!

I'm looking for a way to make the server listen to a range of ports (about 100 ports) and forward all traffic to a different ip:port once a connection is made. Do you think is possible?

[Reply](#)**kaia taylor** *says:*

February 24, 2014 at 6:15 pm

I'm looking for critique/alternate suggestions on this line of thought:

I want Junior, a non-admin user, to reboot a specified list of computers. Ideally, GPO on each target computer could be set up to allow for this, but getting that set up will take quite some time, and I want a simple safe interim solution. I read this post and think what if:

1. Junior authenticates to my tcp server, then
2. Junior chooses from a list of servers I present,

3a. my tcp server sends heads-up email plus reboot command to the servers Junior chose. This is simple but sort of uncomfortable because I have a tcp server run by a privileged id.

3b. Or maybe the server runs as a non-privileged account which is able to send reboot request info to a privileged account that actually does the reboots.

[Reply](#)



---

**Boe Prox** says:

February 24, 2014 at 8:30 pm

Interesting question. I think PowerShell remoting would be better suited for a situation such as this. I say that because you can leverage a custom console session that can limit the number of commands available to your user or build out a proxy function that utilizes the restart-computer and send-mailmessage cmdlets into a single function. You could even have the PSSession run as an administrator account with the necessary rights to perform the actions and have the ACLs in place to make sure only specific accounts can access the session.

That being said, The TCP Server would let the user log onto it, and assuming that the impersonation level is set to 'Delegate' (this would be required if you are attempting to cross over to other servers from the remote system; haven't tested with 'Impersonation' level), the user would still have to have the necessary rights on the server to issue the commands to the other systems regardless of the rights that the user that runs the TCP server. If you wanted the privileged account to run the commands, then you would have to have some sort of check for specific accounts to allow access to run the required commands 'as the privileged account'.

Hope this helps! Let me know if you have any more questions or need more info on what I have mentioned.

[Reply](#)



---

**rjasonmorgan** says:

February 23, 2014 at 11:18 pm

Funny it sounds like you're about to fix the shortcomings in this: <http://gallery.technet.microsoft.com/2d191bcd-3308-4edd-9de2-88dff796b0bc>

The Windows update module From Michal Gajda.

This is really cool stuff!

[Reply](#)

---

**Learn Powershell | Achieve More**

*The Twenty Ten Theme.    Blog at WordPress.com.*