Secrets of PowerShell Remoting

by

Don Jones and Dr. Tobias Weltner

http://PowerShellBooks.com

Foreword

Note This is the August 2012 edition of this guide. You'll find that this guide is periodically updated with additional material, corrections, and expansions. The one, authoritative source for the latest edition is http://PowerShellBooks.com - please check to make sure you've got the latest!

We set out to write this guide because we were getting a lot of questions about some of the trickier bits of PowerShell Remoting, and we wanted to provide an authoritative resource for people seeking to configure Remoting properly for a variety of situations. The guide was started by Don Jones, and Tobias Weltner came aboard to provide a great deal of additional material and improvements. Along the way, numerous members of the PowerShell community contributed their expertise as technical reviewers, suggested additional content to include, and so forth – truly making this a collaborative, group effort. We thank everyone for their continued support, and hope that you find this guide to be useful.

Don Jones and Tobias Weltner,
Principal Authors

Acknowledgements

We'd like to thank Aleksandar Nikolic for his tech-edit of portions of this guide, although we take full responsibility for any remaining errors. We'd also like to thank Steve Larson for the technical corrections he offered to us.

Contents

Remoting Basics

PowerShell v2 introduced a powerful new technology, Remoting, which was refined and expanded upon for PowerShell v3. Based primarily upon standardized protocols and techniques, Remoting is possibly one of the most important aspects of PowerShell: Future Microsoft products will rely upon it almost entirely for administrative communications across the network.

Unfortunately, Remoting is also a complex set of components, and while Microsoft has attempted to provide solid guidance for using it in a variety of scenarios, many administrators still struggle with it. This "mini eBook" is designed to help you better understand what Remoting is, how it works, and – most importantly – how to use it in a variety of different situations.

> **Note** This guide isn't meant to replace the myriad existing books that cover Remoting basics, such as Don's own *Learn Windows PowerShell in a Month of Lunches* (http://MoreLunches.com) or *PowerShell in Depth* (http://PowerShellBooks.com). Instead, this guide supplements those by providing step-by-step instructions for many of the "edge" cases in Remoting, and by explaining some of the more unusual Remoting behaviors and requirements.

What is Remoting?

In essence, Remoting enables you to access remote machines across a network and retrieve data from or execute code on one or many remote computers. This is not a new idea, and in the past a number of different remoting technologies evolved. Some cmdlets have traditionally provided their own limited remoting capabilities while the majority of cmdlets do not support remoting on their own.

With PowerShell remoting, finally there is a generic remoting environment that allows remote execution for literally any command that can run in a local PowerShell. So instead of adding remoting capabilities to every single cmdlet and application, you simply leave it to PowerShell to transfer your PowerShell code to the target computer(s), execute it there, and then marshal back the results to you.

Throughout this eBook, we will focus on PowerShell remoting and not cover non-standard private remoting capabilities built into selected cmdlets.

Examining Remoting Architecture

As shown in figure 1.1, PowerShell's generic Remoting architecture consists of numerous different, interrelated components and elements.

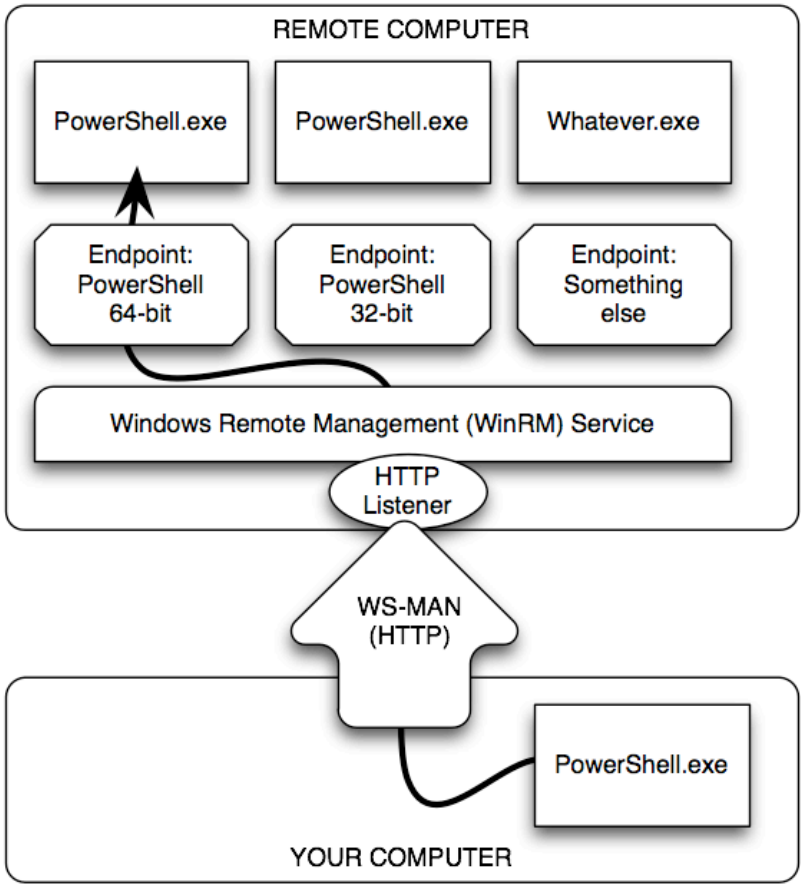


Figure 1.1 The elements and components of PowerShell Remoting

Here's the complete list:

- At the bottom of the figure is your computer, or more properly your *client*. This is where you physically sit, and it's where you'll initiate most of your Remoting activities.

- Your computer will communicate via the WS-MAN, or Web Services for Management, protocol. This is an HTTP(S)-based protocol that can encapsulate a variety of different communications. We've illustrated this as using HTTP, which is Remoting's default configuration, but it could just as easily be HTTPS.
- On the remote computer – the *server*, in the proper terminology, although it could easily be a client operating system like Windows 7 – the Windows Remote Management (WinRM) service runs. This service is configured to have one or more *listeners*. Each listener waits for incoming WS-MAN traffic on a specific port, for a specific protocol (HTTP or HTTPS), and on specific IP addresses (or on all local addresses).
- When a listener receives traffic, the WinRM service looks to see which *endpoint* the traffic is meant for. For our purposes, an endpoint will usually be launching an instance of Windows PowerShell. In PowerShell terms, an endpoint is also called a *session configuration*, because in addition to launching PowerShell, it can auto-load scripts and modules, place restrictions upon what can be done by the connecting user, and so forth.

> **Note** Although we show PowerShell.exe in our diagram, that's for illustration purposes. PowerShell.exe is the PowerShell console application, which wouldn't make sense running in the background on a remote computer. The actual process is Wsmprovhost.exe, which hosts PowerShell in the background for Remoting connections.

As you can see, a single remote computer can easily have dozens or even hundreds of endpoints, each with a different configuration. PowerShell v3 sets up 3 such endpoints by default: One for 32-bit PowerShell (on 64-bit systems), the default PowerShell endpoint (which is 64-bit on x64 systems), and one for PowerShell workflow. Beginning with Server 2008 R2, there is a fourth default endpoint for Server Manager workflow tasks.

Enabling Remoting

Most client versions of Windows (Windows Vista, Windows 7, and so on) do not enable incoming Remoting connections by default. Newer server versions do, but older server versions may not. So your first step with Remoting will usually be to enable it on those computers that you want to receive incoming connections. There are three ways to enable Remoting, and table 1.1 compares what you can achieve with each.

Table 1.1 Comparing the ways of enabling remoting

	Enable-PSRemoting	Group Policy	Manually Step-by-Step
Set WinRM to auto-start and start the service	Yes	Yes	Yes – use Set-Service and Start-Service.

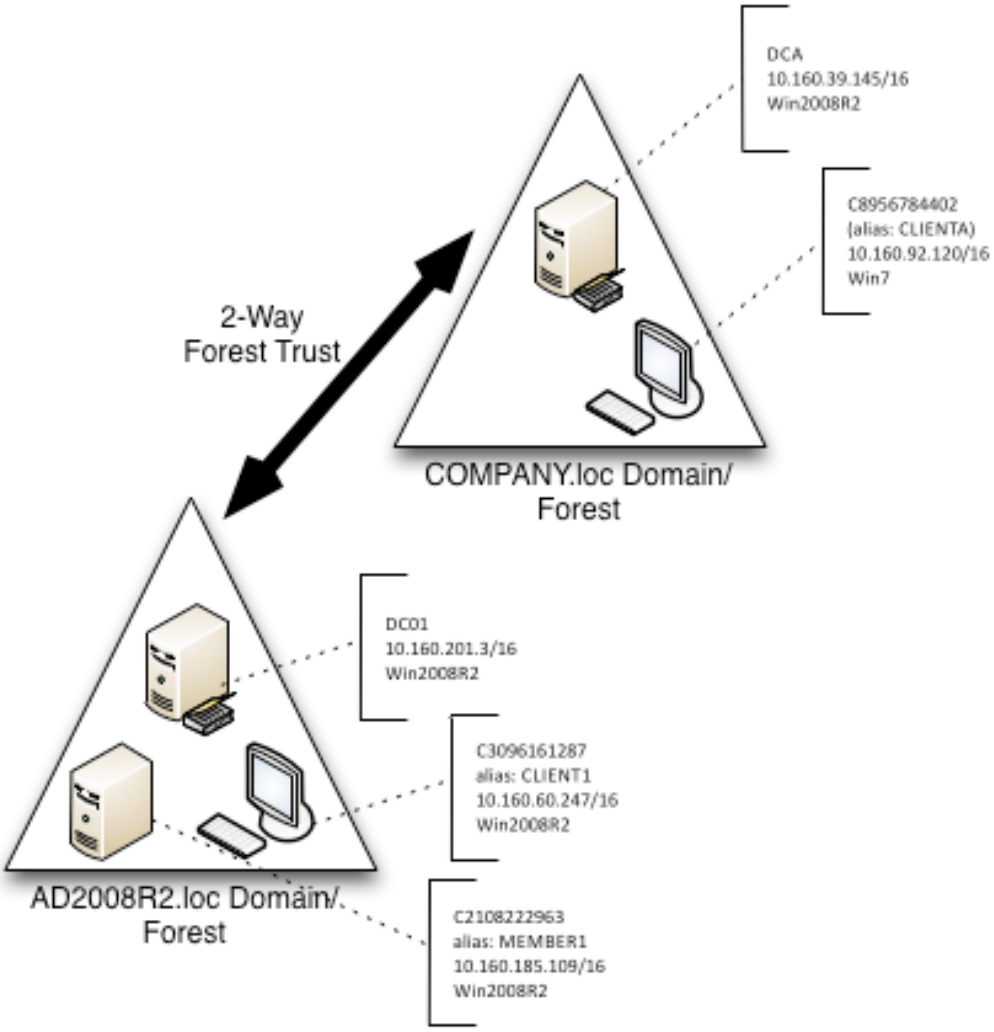
Configure HTTP listener	Yes	You can configure auto-registration of listeners, not create custom listeners	Yes – use WSMAN command-line utility and WSMAN: drive in PowerShell
Configure HTTPS listener	No	No	Yes – use WSMAN command-line utility and WSMAN: drive in PowerShell
Configure endpoints / session configurations	Yes	No	Yes – use PSSessionConfiguration cmdlets
Configure Windows Firewall exception	Yes*	Yes*	Yes* - use Firewall cmdlets or Windows Firewall GUI

> ***Note** Client versions of Windows (Windows Vista, Windows 7, and so on) will not permit firewall exceptions on any network identified as "Public." Networks must either be "Home" or "Work/Domain" in order to permit exceptions. In PowerShell v3, you can run Enable-PSRemoting with the –SkipNetworkProfileCheck switch to avoid this problem.

We'll be enabling Remoting in our test environment by running Enable-PSRemoting. It's quick, easy, and comprehensive; you'll also see most of the manual tasks performed in upcoming sections.

Test Environment

We'll be using a consistent test environment throughout the following sections; this was created on six virtual machines at CloudShare.com, and is configured as shown in figure 1.2.



Some important notes:

- .NET Framework v4 and PowerShell v3 is installed on all computers. Most of what we'll cover also applies to PowerShell v2.
- As shown, most computers have a numeric computer name (C2108222963, and so on); the domain controller for each domain (which is also a DNS server) has CNAME records with easier-to-remember names.
- Each domain controller has a conditional forwarder set up for the other domain, so that

machines in either domain can resolve computer names in the other domain.

- We performed all tasks as a member of the Domain Admins group, unless noted otherwise.

We created a sixth, completely standalone server that isn't in any domain at all. This will be useful for covering some of the non-domain scenarios that you can get into with Remoting.

> **Caution** When opening PowerShell on a computer that has User Account Control (UAC) enabled, make sure you right-click the PowerShell icon and select "Run as Administrator." If the resulting PowerShell window's title bar doesn't specifically say, "Administrator," then you don't have admin privileges. You can check permissions programmatically with this simple line: (whoami /all | select-string S-1-16-12288) -ne $null. In an elevated shell it returns "True", otherwise "False".

Enabling Remoting

We began by running Enable-PSRemoting on all six computers. We took care to ensure that the command ran without error; any errors at this point are a signal that you must stop and resolve the error before attempting to proceed. Figure 1.3 shows the expected output.

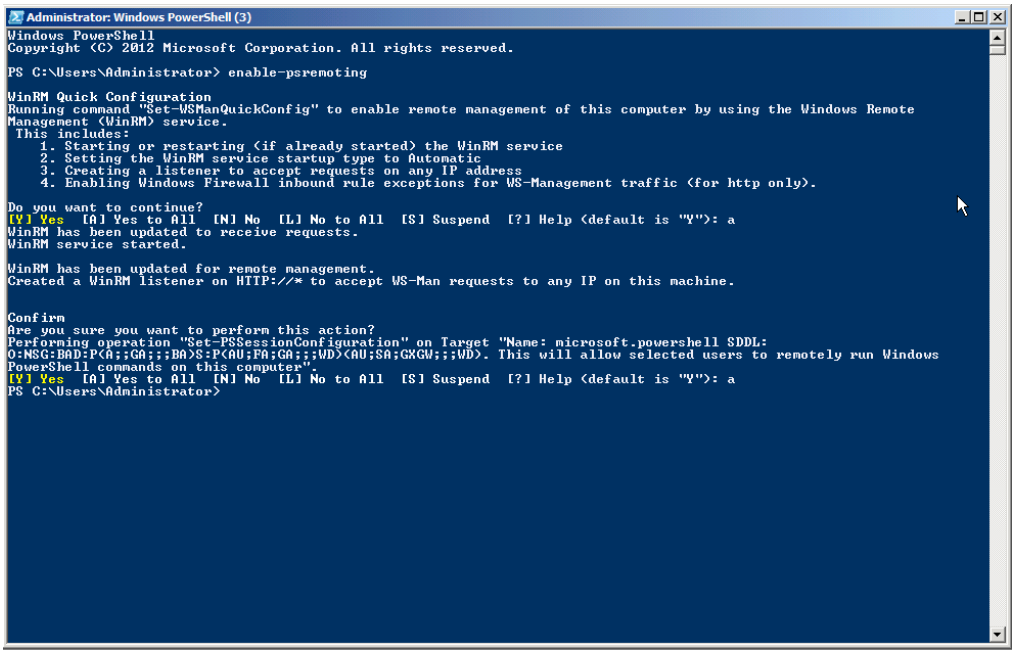


Figure 1.3 Expected output from Enable-PSRemoting

Note You'll notice profligate use of screen shots throughout this guide. It helps ensure that I don't make any typos or copy/paste errors – you're seeing exactly what we typed and ran.

Running Get-PSSessionConfiguration should reveal the three or four endpoints created by Enable-PSRemoting. Figure 1.4 shows the expected output on a server.

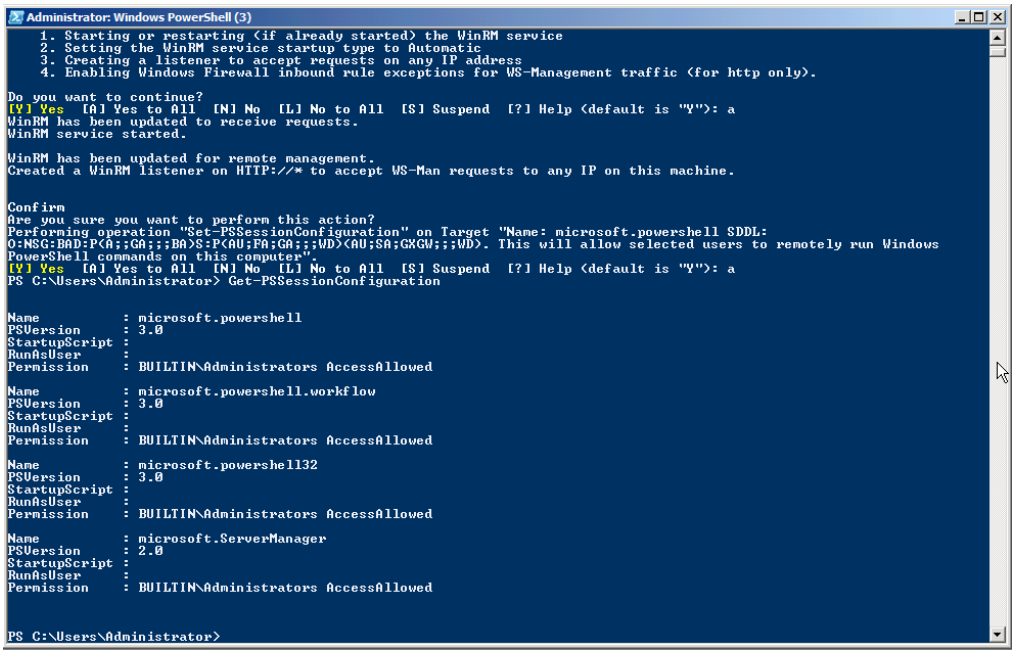
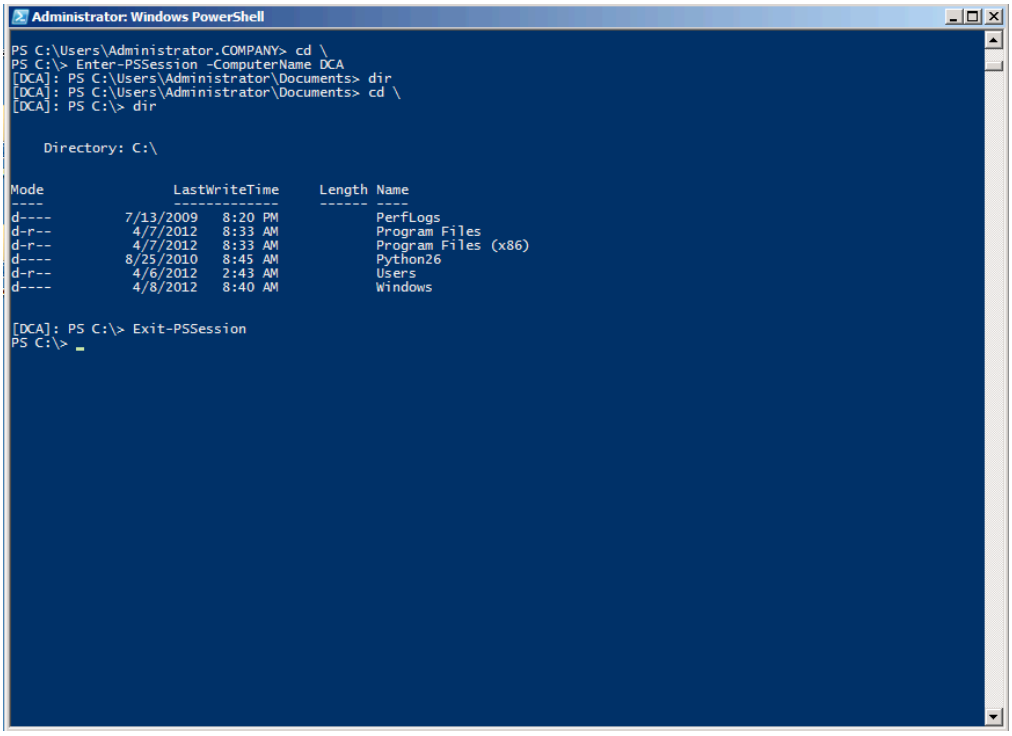


Figure 1.4 Expected output from Get-PSSessionConfiguration

Note Figure 1.4 illustrates that you can expect different endpoints to be configured on different machines. This example was from a Windows Server 2008 R2 computer, which has fewer endpoints than a Windows 2012 machine.

It's worth taking a moment to quickly test the Remoting configuration. For computers that are all part of the same domain, when you're logged on as a Domain Admin from that domain, Remoting should "just work." Quickly check it by attempting to remote from one computer to another using Enter-PSSession. Figure 5 shows the expected output, in which we also ran a quick Dir command and then exited the remote session.



> **Caution** If you're following along in your own test environment, don't proceed until you've confirmed Remoting connectivity between two computers in the same domain. No other scenario needs to work right now; we'll get to them in the upcoming sections.

Core Remoting Tasks

PowerShell provides for two principal Remoting use cases. The first, 1-to-1 Remoting, is similar in nature to the SSH secure shell offered on UNIX and Linux systems. With it, you get a command-line prompt on a single remote computer. The second, 1-to-Many Remoting, enables you to send a command (or a list of commands) in parallel to a set of remote computers. There are also a couple of useful secondary techniques we'll look at.

1-to-1 Remoting

The Enter-PSSession command connects to a remote computer and gives you a command-line prompt on that computer. You can run whatever commands are on that computer, provided you have permission to perform that task. Note that you are not creating an interactive logon session; your connection will be audited as a network logon, just as if you were connecting to the

computer's C$ administrative share. PowerShell will not load or process profile scripts on the remote computer. Any scripts that you choose to run (and this includes importing script modules) will only work if the remote machine's Execution Policy permits it.

```
Enter-PSSession –computerName DC01
```

> **Note** While connected to a remote machine via Enter-PSSession, your prompt changes and displays the name of the remote system in square brackets. If you have customized your prompt, all customizations will be lost because the prompt is now created on the remote system and transferred back to you. All of your interactive keyboard input is sent to the remote machine, and all results are marshaled back to you. This is important to note because you cannot use Enter-PSSession in a script. If you did, the script would still run on your local machine since no code was entered interactively.

1-to-Many Remoting

With this technique, you specify one or more computer names and a command (or a semicolon-separated list of commands); PowerShell sends the commands, via Remoting, to the specified computers. Those computers execute the commands, serialize the results into XML, and transmit the results back to you. Your computer deserializes the XML back into objects, and places them in the pipeline of your PowerShell session. This is accomplished via the Invoke-Command cmdlet.

```
Invoke-Command –computername DC01,CLIENT1 –scriptBlock { Get-Service }
```

If you have a script of commands to run, you can have Invoke-Command read it, transmit the contents to the remote computers, and have them execute those commands.

```
Invoke-Command –computername DC01,CLIENT1 –filePath c:\Scripts\Task.ps1
```

Note that Invoke-Command will, by default, communicate with only 32 computers at once. If you specify more, the extras will queue up, and Invoke-Command will begin processing them as it finishes the first 32. The –ThrottleLimit parameter can raise this limit; the only cost is to your computer, which must have sufficient resources to maintain a unique PowerShell session for each computer you're contacting simultaneously. If you expect to receive large amounts of data from the remote computers, available network bandwidth can be another limiting factor.

Sessions

When you run Enter-PSSession or Invoke-Command and use their –ComputerName parameter, Remoting creates a connection (or *session*), does whatever you've asked it to, and then closes the connection (in the case of an interactive session created with Enter-PSSession, PowerShell

knows you're done when you run Exit-PSSession). There's some overhead involved in that set-up and tear-down, and so PowerShell also offers the option of creating a persistent connection – called a PSSession. You run New-PSSession to create a new, persistent session. Then, rather than using –ComputerName with Enter-PSSession or Invoke-Command, you use their –Session parameter and pass an existing, open PSSession object. That lets the commands re-use the persistent connection you'd previously created.

When you use the -ComputerName parameter and work with adhoc sessions, each time you send a command to a remote machine, there is a significant delay caused by the overhead it takes to create a new session. Since each call to Enter-PSSession or Invoke-Command sets up a new session, you also cannot preserve state. In the example below, the variable $test is lost in the second call:

```
PS> Invoke-Command –computername CLIENT1 –scriptBlock { $test = 1 }

PS> Invoke-Command –computername CLIENT1 –scriptBlock { $test }

PS>
```

When you use persistent sessions, on the other hand, re-connections are much faster, and since you are keeping and reusing sessions, they will preserve state. So here, the second call to Invoke-Command will still be able to access the variable $test that was set up in the first call

```
PS> $Session = New-PSSession -ComputerName CLIENT1

PS> Invoke-Command –Session $Session –scriptBlock { $test = 1 }

PS> Invoke-Command –Session $Session –scriptBlock { $test }

1

PS> Remove-PSSession -Session $Session
```

Various other commands exist to check the session's status and retrieve sessions (Get-PSSession), close them (Remove-PSSession), disconnect and reconnect them (Disconnect-PSSession and Reconnect-PSSession, which are new in PowerShell v3), and so on. In PowerShell v3, you can also pass an open session to Get-Module and Import-Module, enabling you to see the modules listed on a remote computer (via the opened PSSession), or to import a module from a remote computer into your computer for *implicit Remoting*. Review the help on those commands to learn more.

> **Note** Once you use New-PSSession and create your own persistent sessions, it is your responsibility to do housekeeping and close and dispose the session when you are done with them. Until you do that, persistent sessions remain

active, consume resources and may prevent others from connecting. By default, only 10 simultaneous connections to a remote machine are permitted. If you keep too many active sessions, you will easily run into resource limits. This line demonstrates what happens if you try and set up too many simultaneous sessions:

```
PS> 1..10 | Foreach-Object { New-PSSession -ComputerName CLIENT1 }
```

Remoting Returns Deserialized Data

The results you receive from a remote computer have been serialized into XML, and then deserialized on your computer. In essence, the objects placed into your shell's pipeline are static, detached snapshots of what was on the remote computer at the time your command completed. These deserialized objects lack the methods of the originals objects, and instead only offer static properties.

If you need to access methods or change properties, or in other words if you must work with the live objects, simply make sure you do so on the remote side, before the objects get serialized and travel back to the caller. This example uses object methods on the remote side to determine process owners which works just fine:

```
PS> Invoke-Command  -ComputerName CLIENT1 –scriptBlock { Get-WmiObject -Class
Win32_Process | Select-Object Name, { $_.GetOwner().User } }
```

Once the results travel back to you, you can no longer invoke object methods because now you work with "rehydrated" objects that are detached from the live objects and do not contain any methods anymore:

```
PS> Invoke-Command -ComputerName CLIENT1 –scriptBlock { Get-WmiObject -Class
Win32_Process } | Select-Object Name, { $_.GetOwner().User }
```

Serializing and deserializing is relatively expensive. You can optimize speed and resources by making sure that your remote code emits only the data you really need. You could for example use Select-Object and carefully pick the properties you want back rather than serializing and deserializing everything.

Accessing Remote Computers

There are really two scenarios for accessing a remote computer. The difference between those scenarios primarily lies in the answer to one question: Can WinRM identify and authenticate the remote machine?

Obviously, the remote machine needs to know who you are, because it will be executing commands on your behalf. But you need to know who it is, as well. This mutual authentication – e.g., you authenticate each other – is an important security step. It means that when you type SERVER2, you're *really* connecting to the *real* SERVER2, and not some machine *pretending* to be SERVER2. Lots of folks have posted blog articles on how to disable the various authentication checks. Doing so makes Remoting "just work" and gets rid of pesky error messages – but also shuts off security checks and makes it possible for someone to "hijack" or "spoof" your connection and potentially capture sensitive information – like your credentials.

> **Caution** Keep in mind that Remoting involves delegating a credential to the remote computer. You're doing more than just sending a username and password (which doesn't actually happen all of the time): you're giving the remote machine the ability to execute tasks as if you were standing there executing them yourself. An imposter could do a lot of damage with that power. That is why Remoting focuses on mutual authentication – so that imposters can't happen.

In the easiest Remoting scenarios, you're connecting to a machine that's in the same AD domain as yourself, and you're connecting by using its real computer name, as registered with AD. AD handles the mutual authentication and everything works. Things get harder if you need to:

- Connect to a machine in another domain
- Connect to machine that isn't in a domain at all
- Connect via a DNS alias, or via an IP address, rather than via the machine's actual computer name as registered with AD

In these cases, AD can't do mutual authentication, so you have to do it yourself. You have two choices:

- Set up the remote machine to accept HTTPS (rather than HTTP) connections, and equip it with an SSL certificate. The SSL certificate must be issued by a Certification Authority (CA) that your machine trusts; this enables the SSL certificate to provide the mutual authentication WinRM is after.

- Add the remote machine's name (whatever you're specifying, be it a real computer name, an IP address, or a CNAME alias) to your local computer's WinRM TrustedHosts list. Note that this basically disables mutual authentication: You're allowing WinRM to connect to that one identifier (name, IP address, or whatever) without mutual authentication. This opens the possibility for a machine to pretend to be the one you want, so use due caution.

In both cases, you also have to specify a –Credential parameter to your Remoting command, even if you're just specifying the same credential that you're using to run PowerShell. We'll cover both cases in the next two sections.

> **Note** Throughout this guide, we'll use "Remoting command" to generically refer to any command that involves setting up a Remoting connection. Those include (but are not limited to) New-PSSession, Enter-PSSession, Invoke-Command, and so on.

Setting up an HTTPS Listener

This is one of the more complex things you can do with Remoting, and will involve running a lot of external utilities. Sorry – that's just the way it's done! Right now there doesn't seem to be an easy way to do this entirely from within PowerShell, at least not that we've found. Some things, as you'll see, *could* be done through PowerShell, but are more easily done elsewhere – so that's what I've done.

Your first step is to identify the host name that people will use to access your server. This is very, very important, and it isn't necessarily the same as the server's actual computer name. For example, folks accessing "www.ad2008r2.loc" might in fact be hitting a server named "DC01," but the SSL certificate you'll create must be issued to host name "www.ad2008r2.loc" because that's what people will be typing. So, the certificate name needs to match whatever name people will be typing to get to the machine – even if that's different from its true computer name. Got that?

> **Note** As the above implies, part of setting up an HTTPS listener is obtaining an SSL certificate. I'll be using a public Certification Authority (CA) named DigiCert.com. You could also use an internal PKI, if your organization has one. I don't recommend using MakeCert.exe, since such a certificate can't be implicitly trusted by the machines attempting to connect. I realize that every blog in the universe tells you to use MakeCert.exe to make a local self-signed certificate. Yes, it's easy – but it's *wrong*. Using it requires you to shut off most of WinRM's security – so why bother with SSL if you plan to shut off most of its security features?

You need to make sure you know the full name used to connect to a computer, too. If people will have to type "dc01.ad2008r2.loc," then that's what goes into the certificate. If they'll simply need to provide "dca," and know that DNS can resolve that to an IP address, then "dca"

is what goes into the certificate. We're creating a certificate that just says "dca" and we'll make sure our computers can resolve that to an IP address.

Creating a Certificate Request

Unlike IIS, PowerShell doesn't offer a friendly, graphical way of creating a Certificate Request (CSR). Or, in fact, any way at all to do so. So, go to http://DigiCert.com/util and download their certificate utility (it's free). Figure 2.1 shows the utility. Note the warning message.

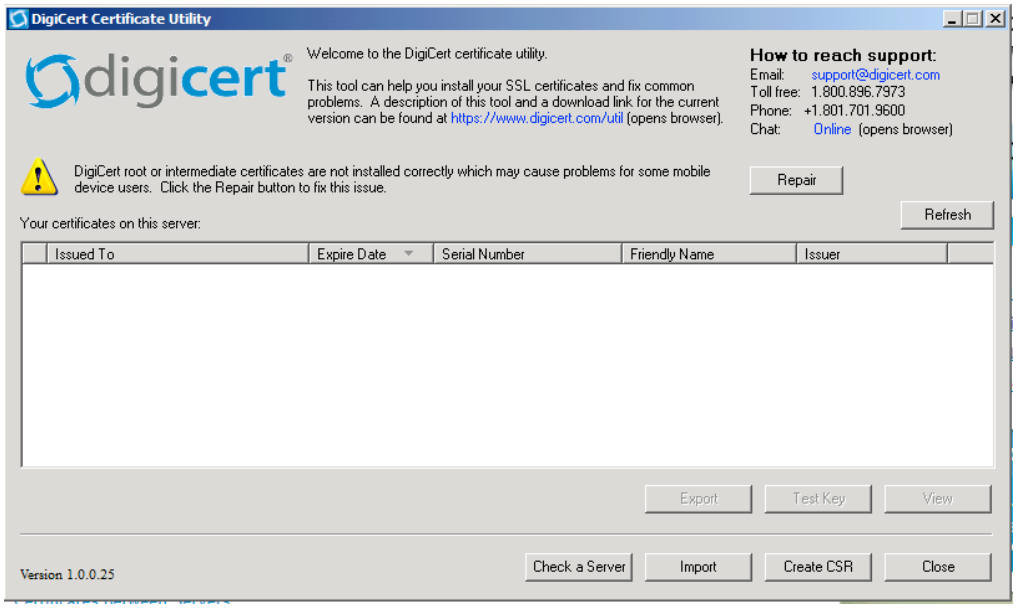
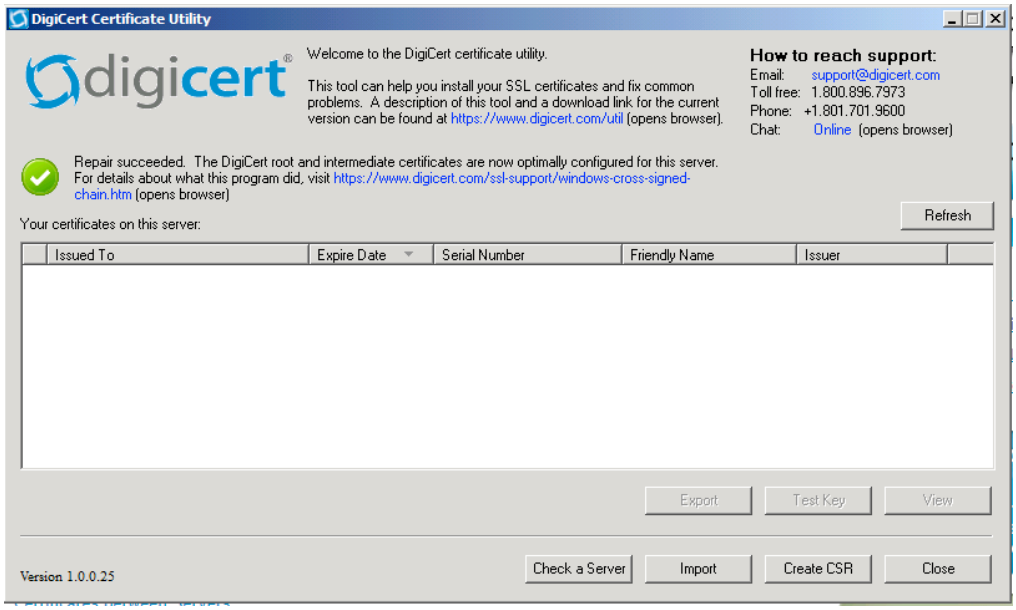
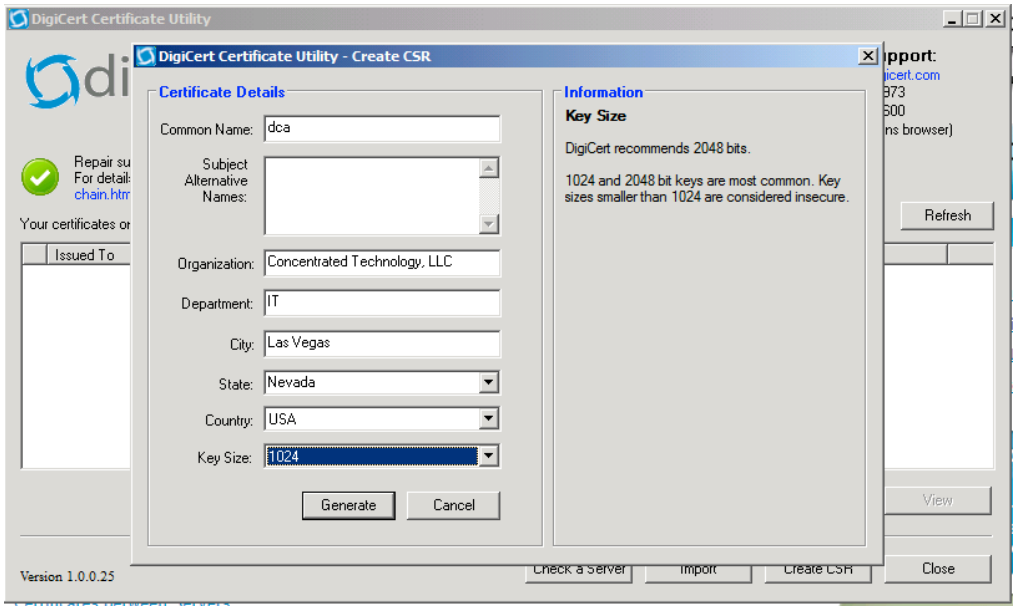


Figure 2.1 Launching DigiCertUtil.exe

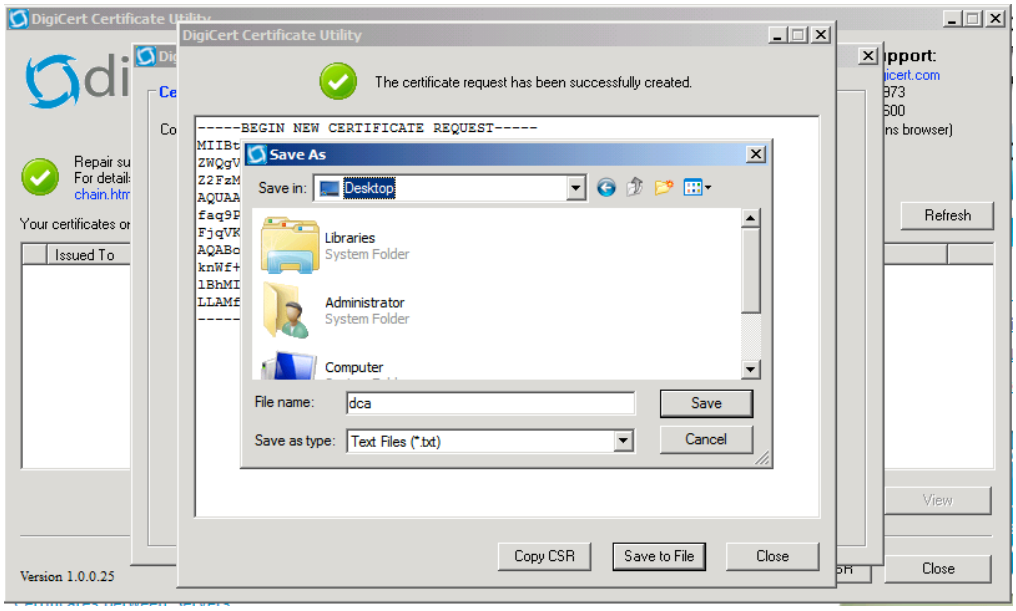
You only need to worry about this warning if you plan to acquire your certificate from the DigiCert CA; click the Repair button to install their intermediate certificates on your computer, enabling their certificate to be trusted and used. Figure 2.2 shows the result of doing so. Again, if you plan to take the eventual Certificate Request (CSR) to a different CA, don't worry about the Repair button or the warning message.



Click "Create CSR." As shown in figure 2.3, fill in the information about your organization. This needs to be exact: The "Common Name" is exactly what people will type to access the computer on which this SSL certificate will be installed. That might be "dca," in our case, or "dc01.ad20082.loc" if a fully qualified name is needed, and so on. Your company name also needs to be accurate: Most CAs will verify this information.

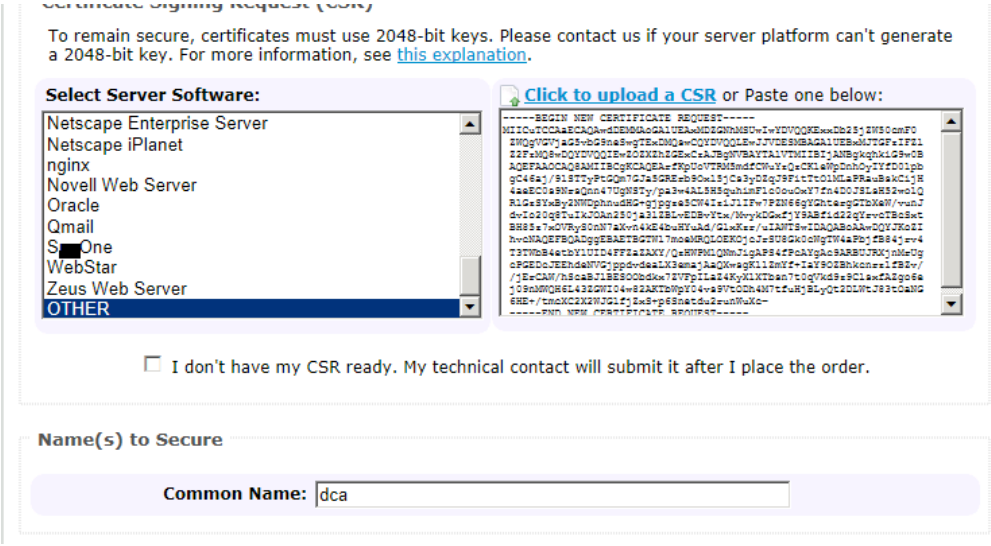


We usually save the CSR in a text file, as shown in figure 2.4. You can also just copy it to the Clipboard in many cases. When you head to your CA, make sure you're requesting an SSL ("Web Server," in some cases) certificate. An e-mail certificate or other type won't work.



Next, take that CSR to your CA and order your certificate. This will look something like figure 2.5 if you're using DigiCert; it'll obviously be different with another CA, with an internal PKI, and so forth. Note that with most commercial CAs you'll have to select the type of Web server you're using; choose "Other," if that's an option, or "IIS" if not.

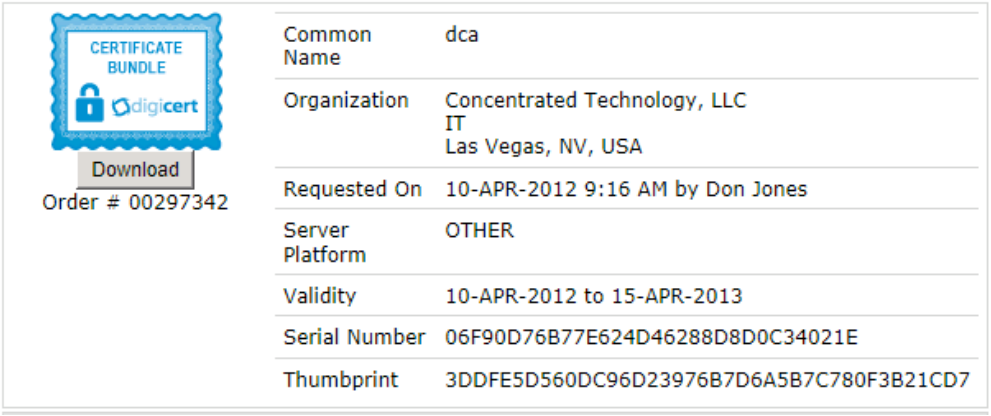
> **Note** Using the MakeCert.exe utility from the Windows SDK will generate a local certificate that only your machine will trust. This isn't useful. Folks tell you to do this in various blog posts because it's quick and easy; they also tell you to disable various security checks so that the inherently-useless certificate will work. It's a waste of time. You're getting encryption, but you've no assurance that the remote machine is the one you intended to connect to in the first place. If someone's hijacking your information, who cares if it was encrypted before you sent it to them?



Caution Note the warning message in figure 2.5 that my CSR needs to be generated with a 2048-bit key. DigiCert's utility offered me that, or 1024-bit. Many CAs will have a high-bit requirement; make sure your CSR complies with what they need. Also notice that this is a Web server certificate we're applying for; as we wrote earlier, it's the only kind of certificate that will work.

Eventually, the CA will issue your certificate. Figure 2.6 shows where we went to download it. We chose to download all certificates; we wanted to ensure we had a copy of the CA's root certificate, in case we needed to configure another machine to trust that root.

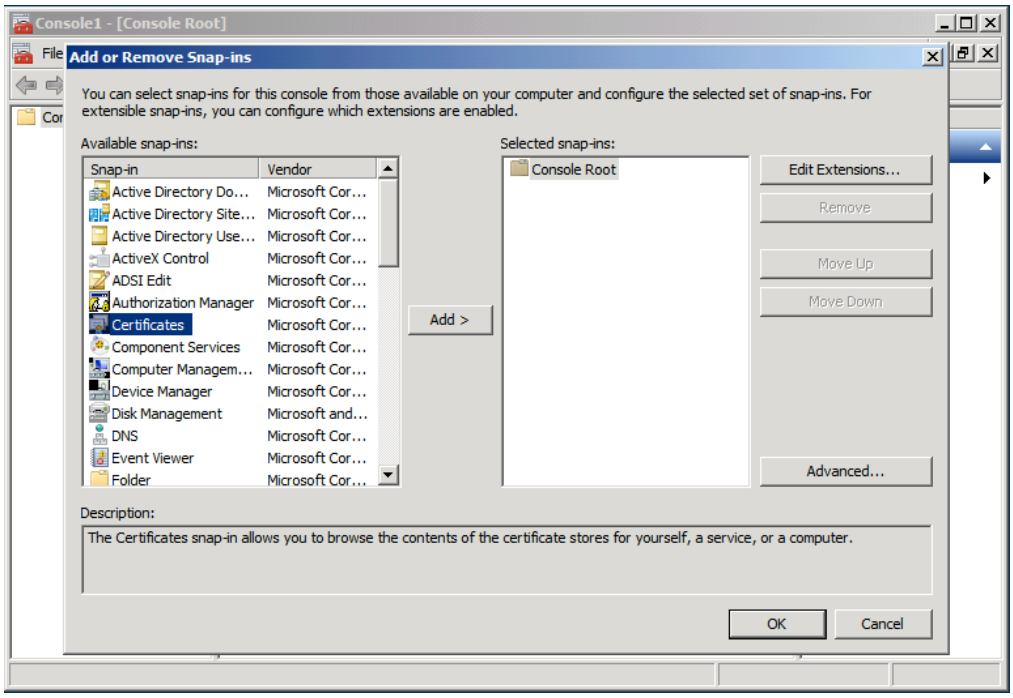
Tip The trick with digital certificates is that the machine using them, and any machines they will be presented to, need to trust the CA that issued the certificate. That's why you download the CA root certificate: so you can install it on the machines that need to trust the CA. In a large environment, this can be done via Group Policy, if desired.



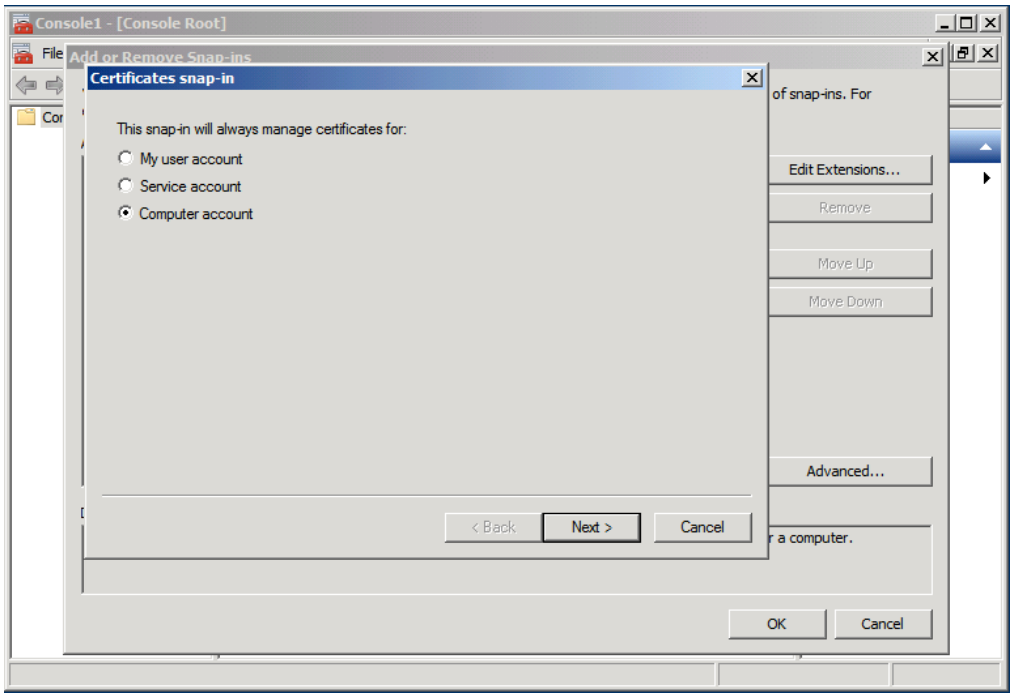
Make sure you back up the certificate files! Even though most CAs will re-issue them as needed, it's far easier to have a handy backup, even on a USB flash drive.

Installing the Certificate

Don't try to double-click the certificate file to install it. Doing so will install it into your user account's certificate store; you need it in your computer's certificate store instead. To install the certificate, open a new Microsoft Management Console (mmc.exe), select Add/Remove Snap-ins, and add the Certificates snap-in, as shown in figure 2.7.

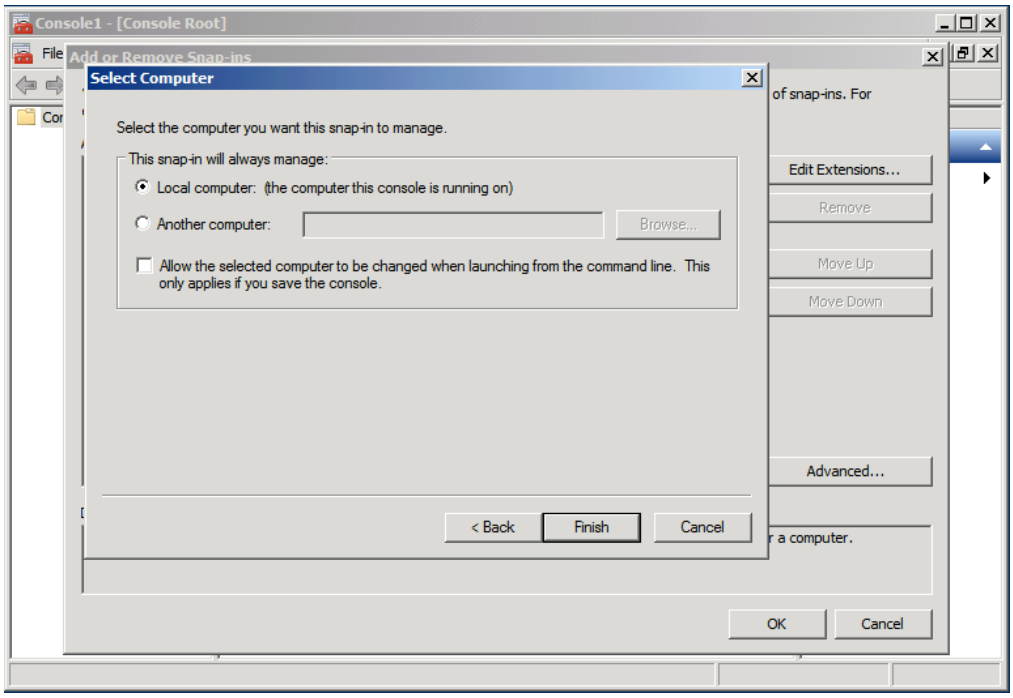


As shown in figure 2.8, focus the snap-in on the Computer account.

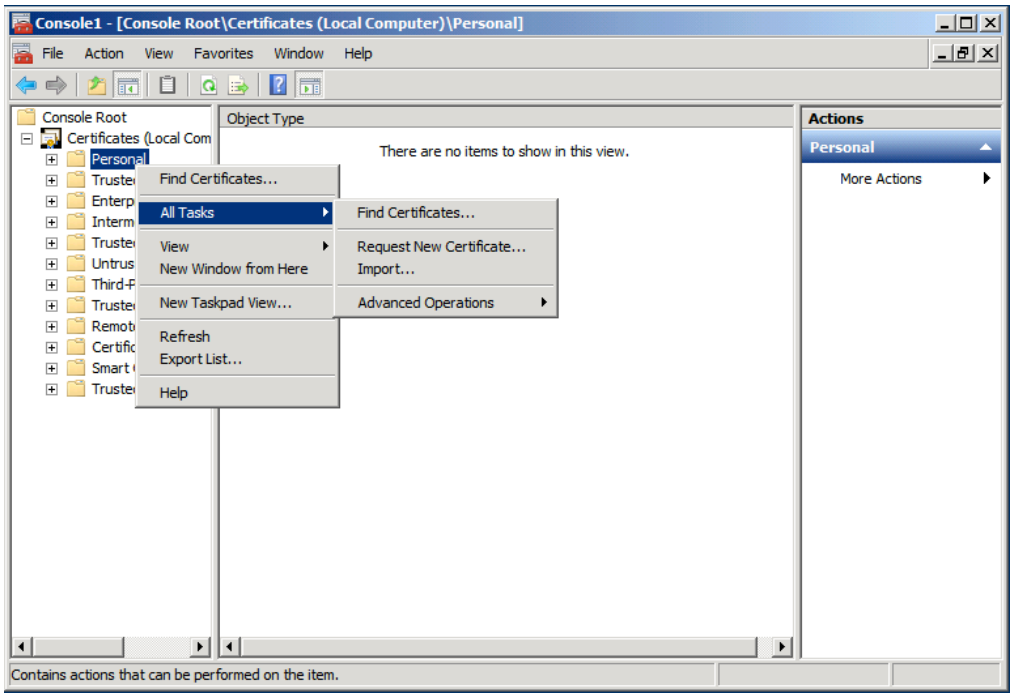


Next, as shown in figure 2.9, focus on the local computer. Of course, if you're installing a certificate onto a remote computer, focus on that computer instead. This is a good way to get a certificate installed onto a GUI-less Server Core installation of Windows, for example.

> **Note** We wish I could show you a way to do all of this from within PowerShell. But we couldn't find one that didn't involve a jillion more, and more complex, steps. Since this hopefully isn't something you'll have to do often, or automate a lot, the GUI is easier and should suffice.

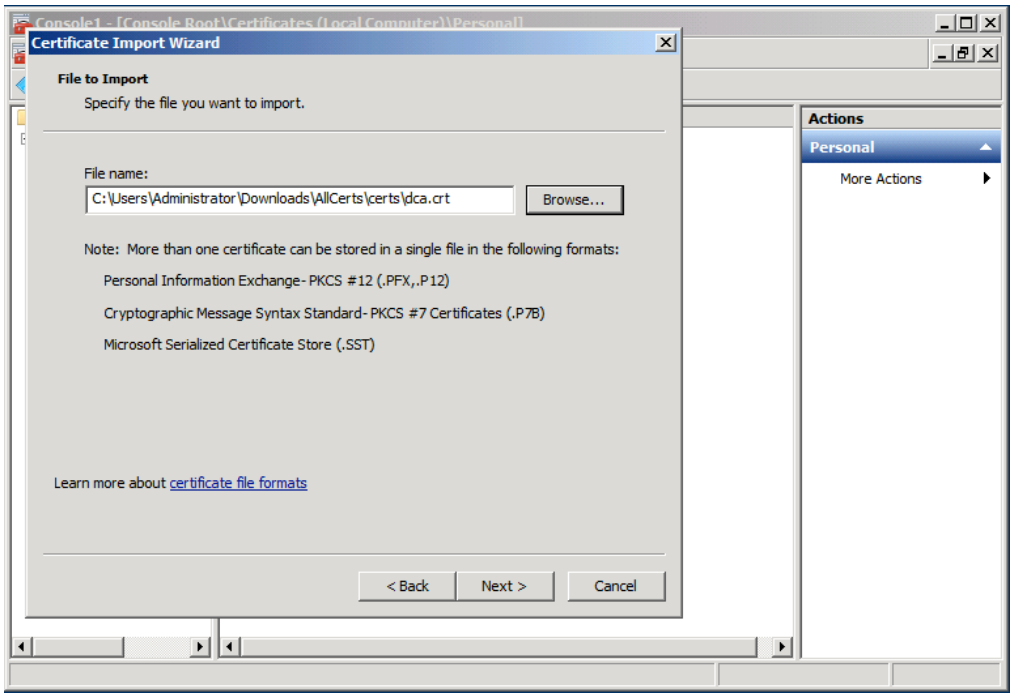


With the snap-in loaded, as shown in figure 2.10, right-click the "Personal" store and select "Import."

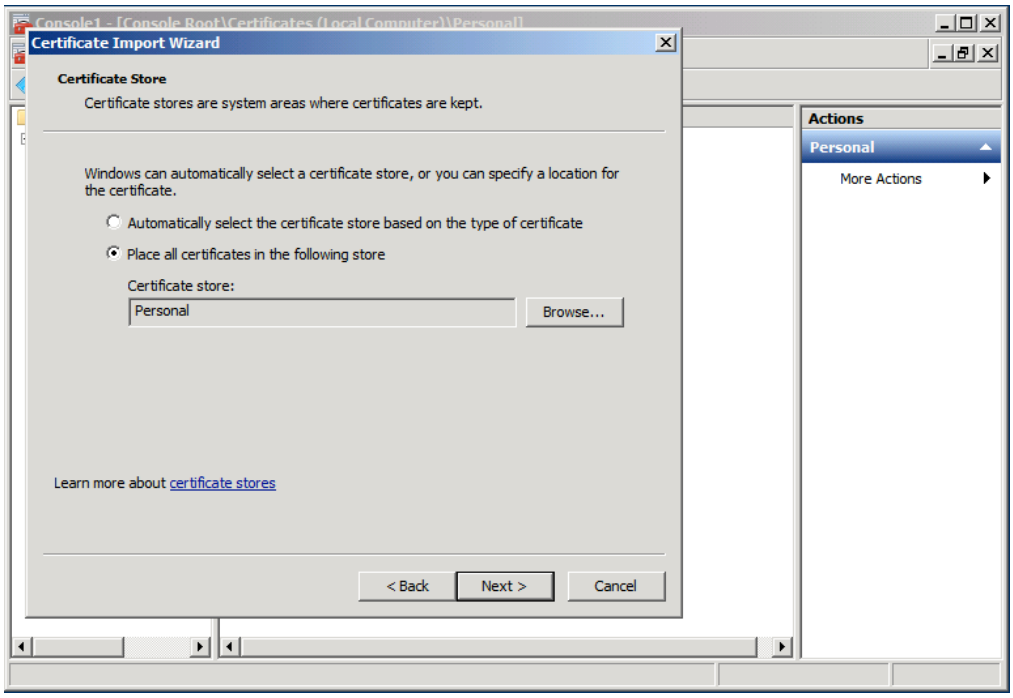


As shown in figure 2.11, browse to the certificate file that you downloaded from your CA. Then, click Next.

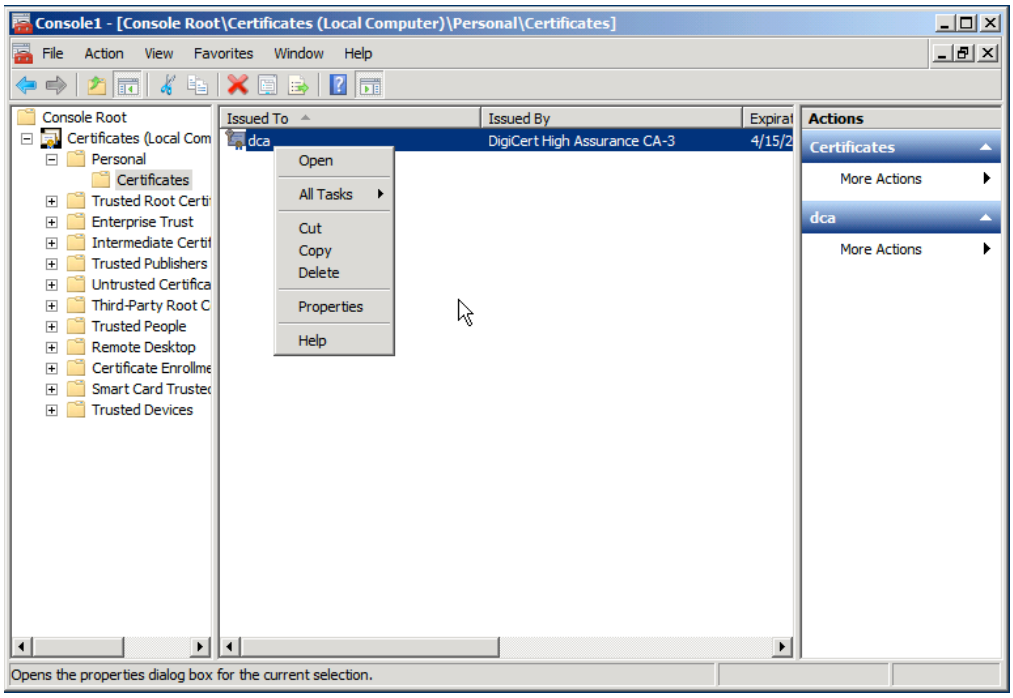
> **Caution** If you downloaded multiple certificates – perhaps the CA's root certificates along with the one issued to you – make sure you're importing the SSL certificate that was issued to you. If there's any confusion, **STOP**. Go back to your CA and download just YOUR certificate, so that you'll know which one to import. Don't experiment, here – **you need to get this right the first time.**



As shown in figure 2.12, ensure that the certificate will be placed into the Personal store.

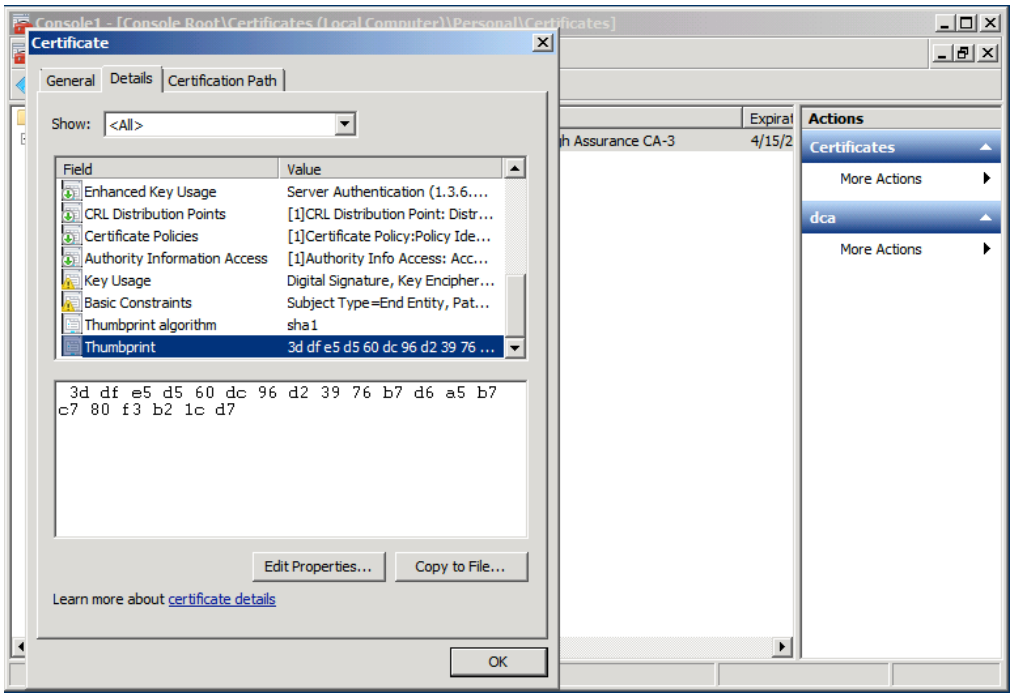


As shown in figure 2.13, double-click the certificate to open it. Or, right-click and select Open. Do not select Properties – that won't get you the information you need.



Finally, as shown in figure 2.14, select the certificate's thumbprint. You'll need to either write this down, or copy it to your Clipboard. This is how WinRM will identify the certificate you want to use.

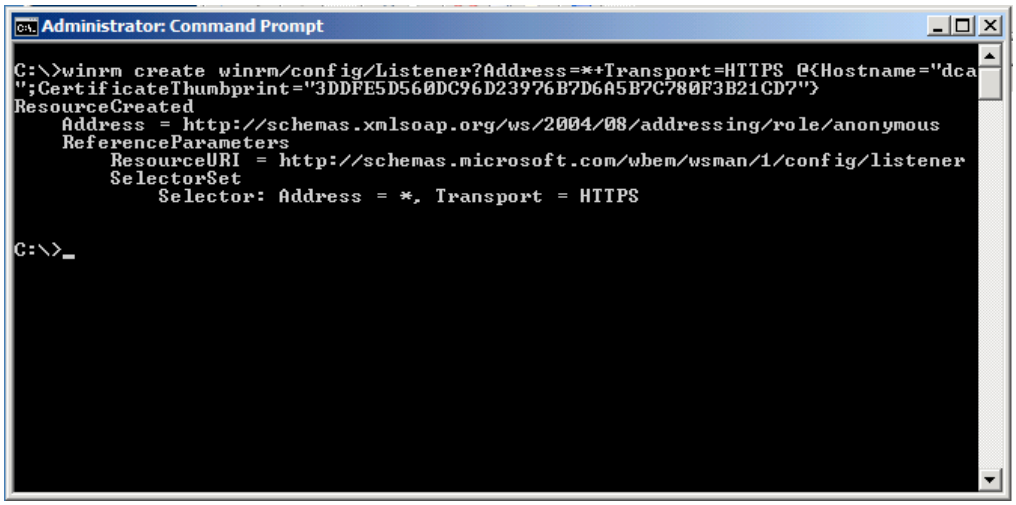
> **Note** It's possible to list your certificate in PowerShell's CERT: drive, which will make the thumbprint a bit easier to copy to the Clipboard. In PowerShell, run `Dir CERT:\LocalMachine\My` and read carefully to make sure you select the right certificate. If the entire thumbprint isn't displayed, run `Dir CERT:\LocalMachine\My | FL *` instead.



Setting up the HTTPS Listener

These next steps will be accomplished in the Cmd.exe shell, not in PowerShell. The command-line utility's syntax requires significant tweaking and escaping in PowerShell, and it's a lot easier to type and understand in the older Cmd.exe shell (which is where the utility has to run anyway; running it in PowerShell would just launch Cmd.exe behind the scenes).

As shown in figure 2.15, run the following command:




```
Winrm create winrm/config/Listener?Address=*+Transport=HTTPS
@{Hostname="xxx";CertificateThumbprint="yyy"}
```

There are two or three pieces of information you'll need to place into this command:

- In place of *, you can put an individual IP address. Using * will have the listener listen to all local IP addresses.
- In place of xxx, put the exact computer name that the certificate was issued to. If that includes a domain name (such as dc01.ad2008r2.loc), put that. Whatever's in the certificate must go here, or you'll get a CN mismatch error. Our certificate was issued to "dca," so I put "dca."
- In place of yyy, put the exact certificate thumbprint that you copied earlier. It's okay if this contains spaces.

That's all you should need to do in order to get the listener working.

> **Note** We had the Windows Firewall disabled on this server, so we didn't need to create an exception. The exception isn't created automatically, so if you have any firewall enabled on your computer, you'll need to manually create the exception for port 5986.
>
> You can also run an equivalent PowerShell command to accomplish this task:

```
New-WSManInstance winrm/config/Listener –SelectorSet @{Address='*';
Transport='HTTPS'} –ValueSet @{HostName='xxx';CertificateThumbprint='yyy'}
```

In that example, "xxx" and "yyy" get replaced just as they did in the previous example.

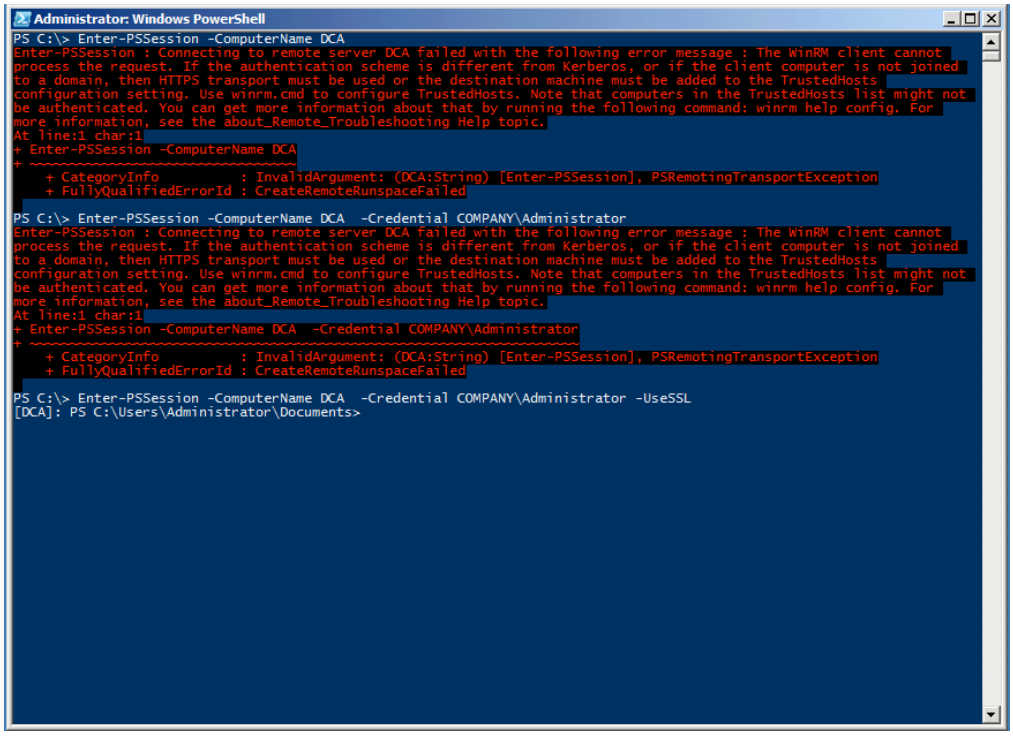
Testing the HTTPS Listener

I tested this from the standalone C3925954503 computer, attempting to reach the DCA domain controller in COMPANY.loc. I configured C3925954503 with a HOSTS file, so that it could resolve the hostname DCA to the correct IP address without needing DNS. I was sure to run:

```
Ipconfig /flushdns
```

Which ensured that the HOSTS file was read into the DNS name cache. The results are in figure 2.16. Note that I can't access DCA by using its IP address directly, because the SSL certificate doesn't contain an IP address. The SSL certificate was issued to "dca," so we need to be able to access the computer by typing "dca" as the computer name. Using the HOSTS file will let Windows resolve that to an IP address.

> **Note** Remember, there's two things going on here. Windows needs to be able to resolve the name to an IP address, which is what the HOSTS file accomplishes, in order to make a physical connection. But WinRM needs mutual authentication, which means whatever we typed in the –ComputerName parameter needs to match what's in the SSL certificate. That's why we couldn't just provide an IP address to the command – it would have worked for the connection, but not the authentication.



We started with this:

```
Enter-PSSession –computerName DCA
```

Didn't work – which I expected. Then we tried this:

```
Enter-PSSession –computerName DCA –credential COMPANY\Administrator
```

We provided a valid password for the Administrator account, but as expected the command didn't work. Finally:

```
Enter-PSSession –computerName DCA –credential COMPANY\Administrator -UseSSL
```

Again providing a valid password, we were rewarded with the remote prompt we expected. It worked! This fulfills the two conditions we specified earlier: We're using an HTTPS-secured connection *and* providing a credential. Both conditions are required because the computer isn't in my domain (since in this case the source computer isn't even in a domain). As a refresher, figure 2.17 shows, in green, the connection we created and used.

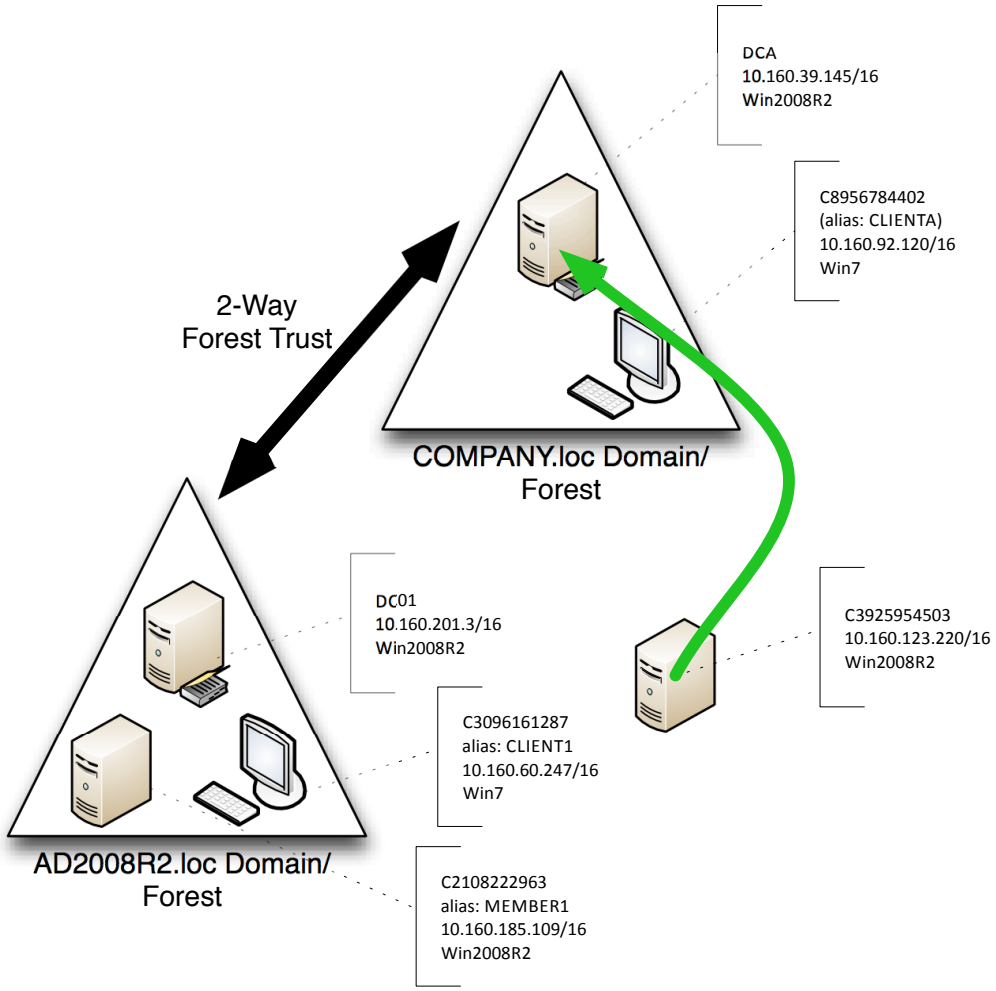


Figure 2.17 The connection used for the HTTPS listener test

Modifications

There are two modifications you can make to a connection, whether using Invoke-Command, Enter-PSSession, or some other Remoting command, which relate to HTTPS listeners. These are created as part of a session option object.

- -SkipCACheck causes WinRM to not worry about whether the SSL certificate was issued by a trusted CA or not. However, untrusted CAs may in fact be untrust*worthy*! A poor CA might issue a certificate to a bogus computer, leading you to believe you're connecting to the right machine when in fact you're connecting to an imposter. This is risky, so use it with caution.

- -SkipCNCheck causes WinRM to not worry about whether the SSL certificate on the remote machine was actually issued for that machine or not. Again, this is a great way to find yourself connected to an imposter. Half the point of SSL is mutual authentication, and this parameter disables that half.

Using either or both of these options will still enable SSL encryption on the connection – but you'll have defeated the other essential purpose of SSL, which is mutual authentication by means of a trusted intermediate authority.

To create and use a session object that includes both of these parameters:

```
$option = New-PSSessionOption –SkipCACheck –SkipCNCheck

Enter-PSSession –computerName DCA –sessionOption $option
                -credential COMPANY\Administrator –useSSL
```

Caution Yes, this is an easy way to make annoying error messages go away. But those errors are trying to warn you of a potential problem and protect you from potential security risks that are very real, and which are very much in use by modern attackers.

Modifying the TrustedHosts List

As I mentioned earlier, using SSL is only one option for connecting to a computer for which mutual authentication isn't possible. The other option is to selectively disable the need for mutual authentication by providing your computer with a list of "trusted hosts." In other words, you're telling your computer, "if I try to access SERVER1 [for example], don't bother mutually authenticating. I know that SERVER1 can't possibly be spoofed or impersonated, so I'm taking that burden off of your shoulders."

Figure 2.18 illustrates the connection we'll be attempting.

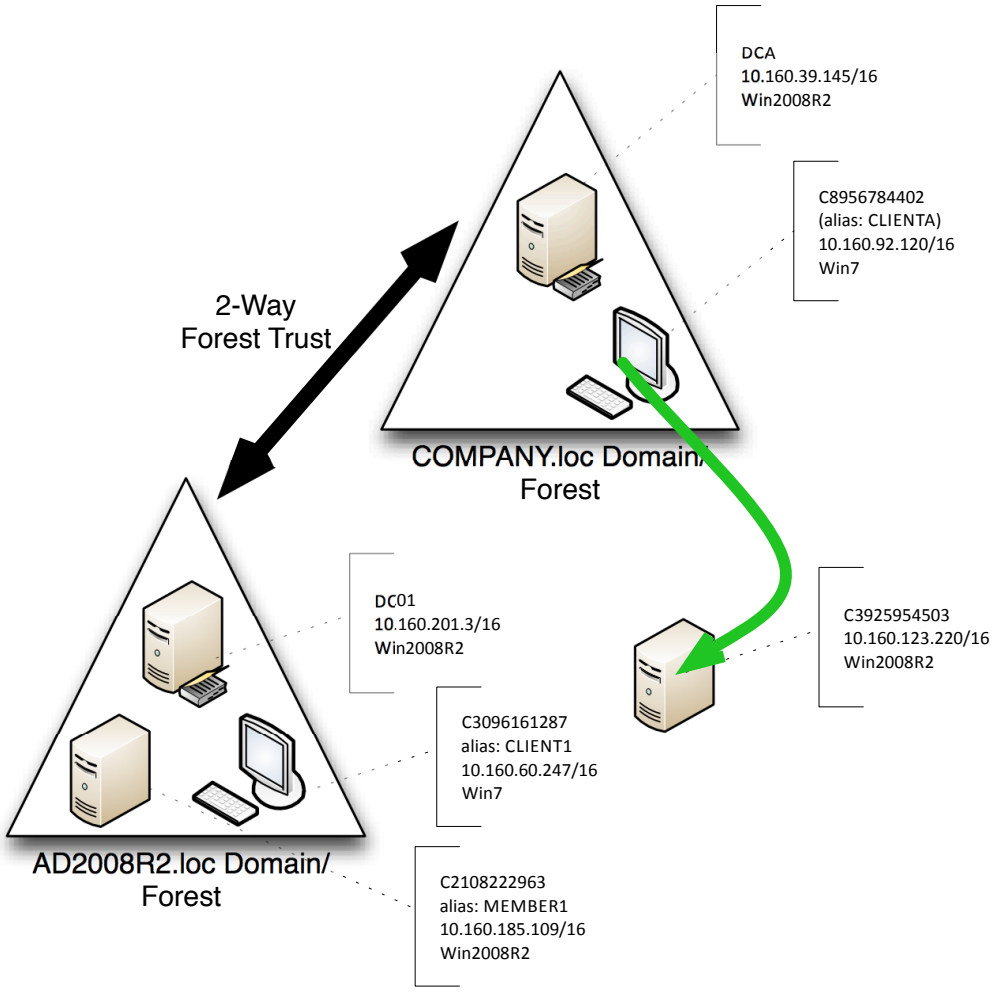
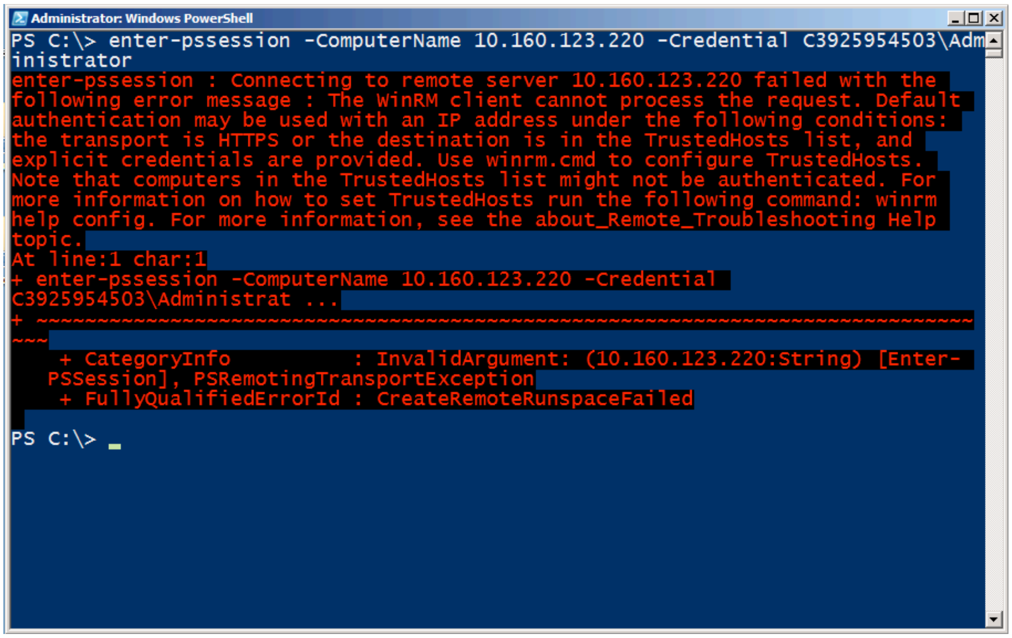


Figure 2.18 The TrustedHosts connection test

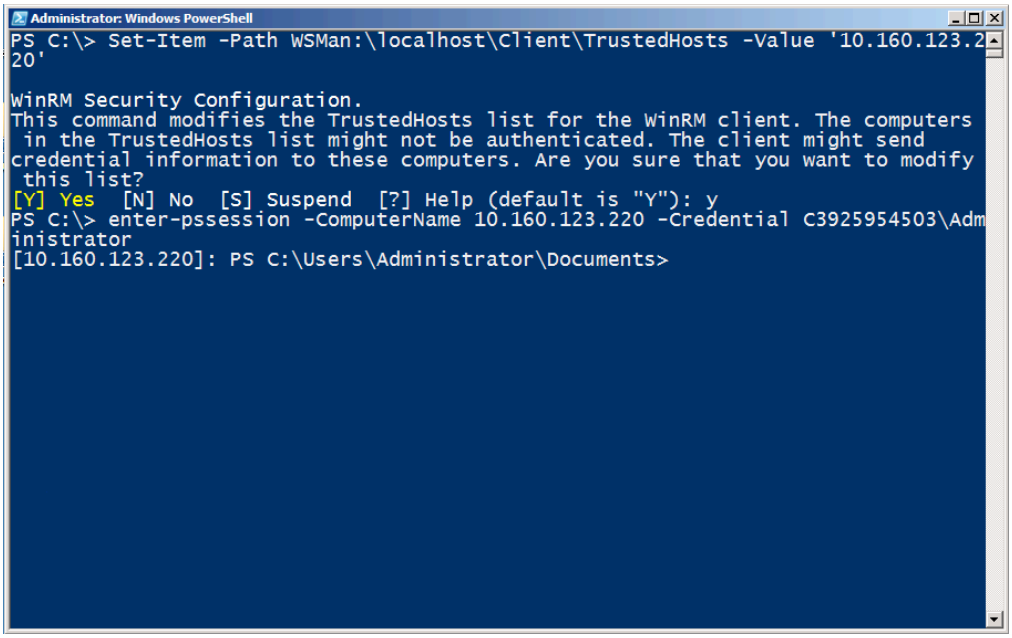
Beginning on CLIENTA, with a completely default Remoting configuration, we'll attempt to connect to C3925954503, which also has a completely default Remoting configuration. Figure 2.19 shows the result. Note that I'm connecting via IP address, rather than hostname; our client has no way of resolving the computer's name to an IP address, and for this test we'd rather not modify my local HOSTS file.



This is what we expected: The error message is clear that we can't use an IP address (or a host name for a non-domain computer, although the error doesn't say so) unless we either use HTTPS and a credential, or add the computer to my TrustedHosts list and use a credential. We'll choose the latter this time; figure 2.20 shows the command we need to run. If we'd wanted to connect via the computer's name (C3925954503) instead of its IP address, we'd have added that computer name to the TrustedHosts list (It'd be our responsibility to ensure my computer could somehow resolve that computer name to an IP address to make the physical connection).

This is another case where many blogs will advise just putting "*" in the TrustedHosts list. Really? There's no chance any computer, ever, anywhere, could be impersonated or spoofed? We prefer adding a limited, controlled set of host names or IP addresses. Use a comma-separated list; it's okay to use wildcards along with some other characters (like a domain name, such as *.COMPANY.loc), to allow a wide, but not unlimited, range of hosts. Figure 2.21 shows the successful connection.

> **Tip** Use the –Concatenate parameter of Set-Item to add your new value to any existing ones, rather than overwriting them.



Managing the TrustedHosts list is probably the easiest way to connect to a computer that can't offer mutual authentication, provided you're absolutely certain that spoofing or impersonation isn't a possibility. On an intranet, for example, where you already exercise good security practices, impersonation may be a remote chance, and you can add an IP address range or host name range using wildcards.

Connecting Across Domains

Figure 2.22 illustrates the next connection we'll try to make, which is between two computers in different, trusted and trusting, forests.

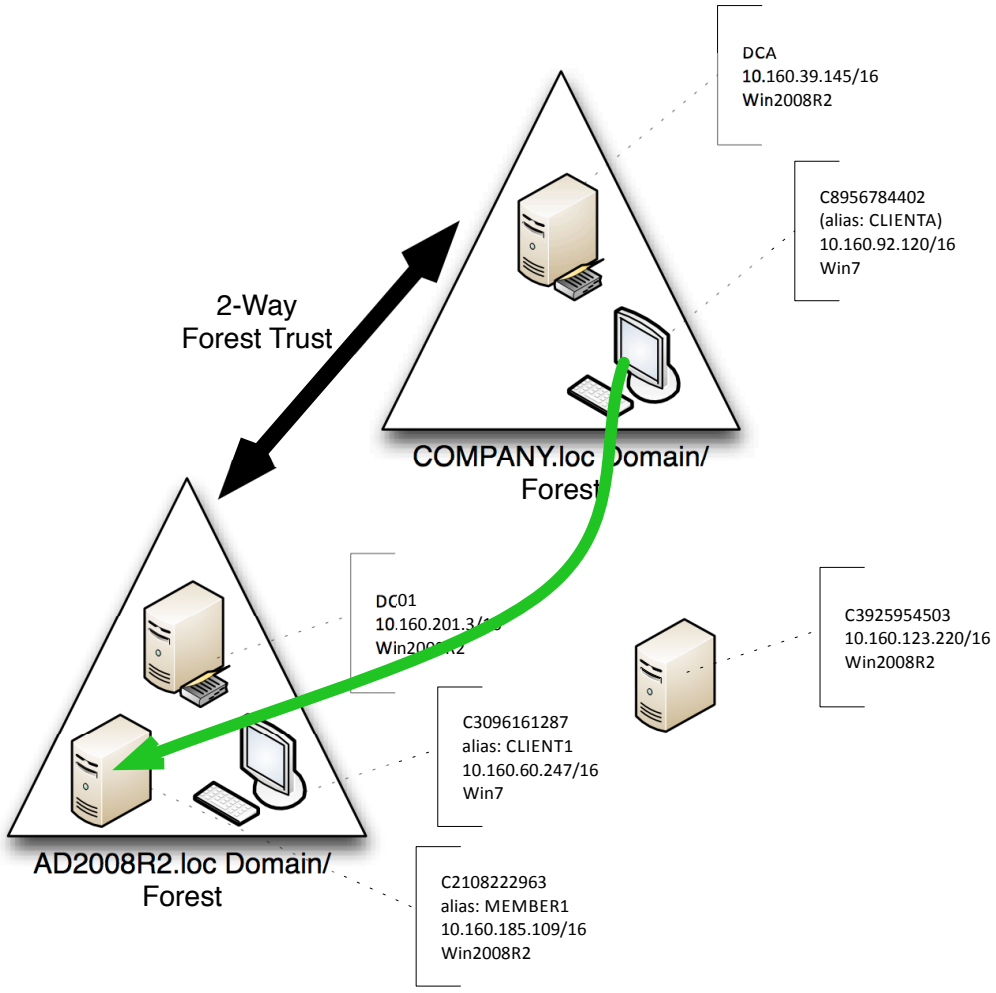
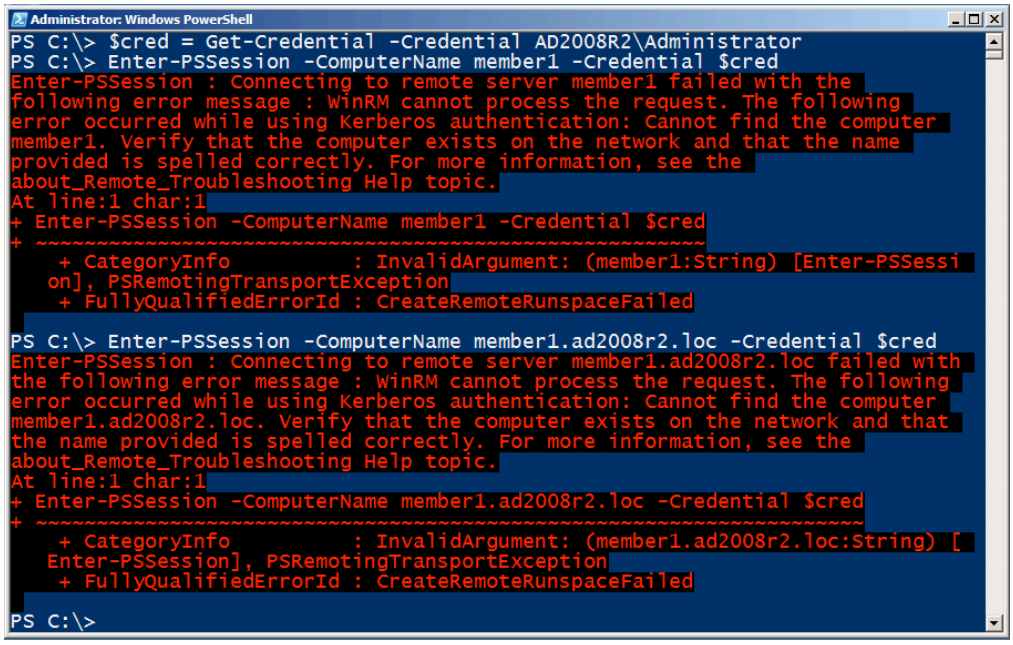
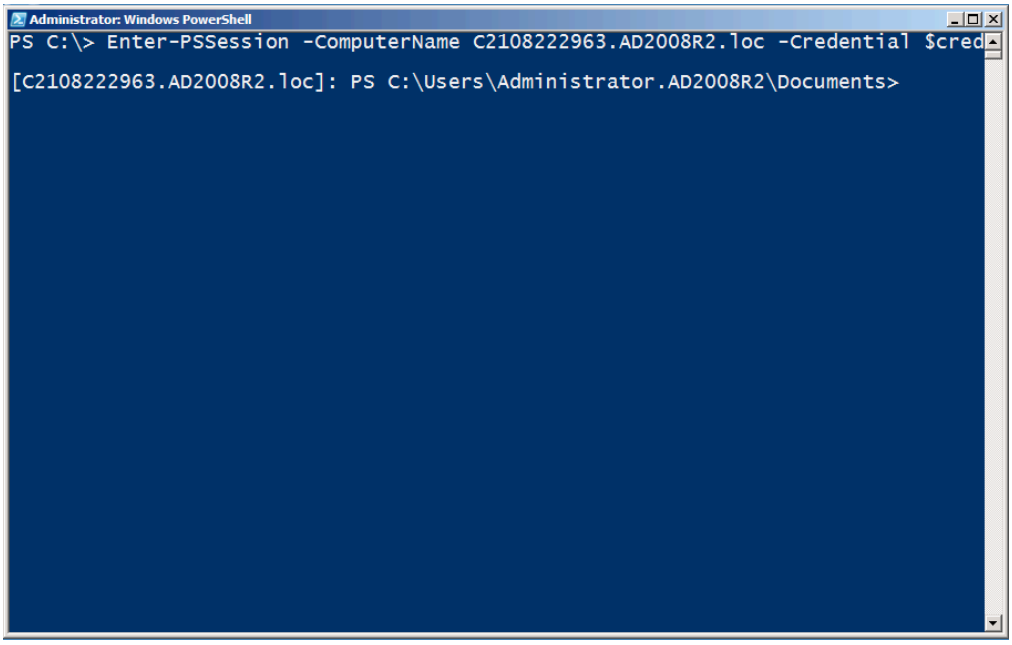


Figure 2.22 Connection for the cross-domain test

Our first test is in figure 2.23. Notice that we're creating a reusable credential in the variable $cred, so that we don't keep having to re-type the password as we try this. However, the results of the Remoting test still aren't successful.



The problem? We're using a CNAME alias (MEMBER1), not the computer's real host name (C2108222963). While WinRM can use a CNAME to resolve a name to an IP address for the physical connection, it can't use the CNAME alias to look the computer up in AD, because AD doesn't use the CNAME record (even in an AD-integrated DNS zone). As shown in figure 2.24, the solution is to use the computer's real host name.



What if you *need* to use an IP address or CNAME alias to connect? Then you'll have to fall back to the TrustedHosts list or an HTTPS listener, exactly as if you were connecting to a non-domain computer. Essentially, if you can't use the computer's real host name, as listed in AD, then you can't rely on the domain to shortcut the whole authentication process.

Administrators from Other Domains

There's a quirk in Windows that tends to strip the Administrator account token for administrator accounts coming in from other domains, meaning they end up running under standard user privileges – which often isn't sufficient. In the target domain, you need to change that behavior.

To do so, run this on the target computer (type this all in one line and then hit Enter):

```
New-ItemProperty –Name LocalAccountTokenFilterPolicy
 -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\
 Policies\System –PropertyType Dword –Value 1
```

That should fix the problem. Note that this does disable User Account Control (UAC) on the machine where you ran it, so make sure that's okay with you before doing so.

The Second Hop

One default limitation with Remoting is often referred to as *the second hop*. Figure 2.25 illustrates the basic problem: You can make a Remoting connection from one host to another (the green line), but going from that second host to a third (the red line) is simply disallowed. This "second hop" doesn't work because, by default, Remoting can't delegate your credential a second time. This is even a problem if you make the first hop and subsequently try to access any network resource that requires authentication. For example, if you remote into another computer, and then ask that computer to access something on an authenticated file share, the operation fails.

The following configuration changes are needed to enable the second hop:

> **Note** This only works on Windows Vista, Windows Server 2008, and later versions of Windows. It won't work on Windows XP or Windows Server 2003 or earlier versions.

- CredSSP must be enabled on your originating computer and the intermediate server you connect to. In PowerShell, on your originating computer, run:

```
Set-Item WSMAN:\localhost\client\auth\credssp –value $true
```

- On your intermediate server(s), you make a similar change to the above, but in a different section of the configuration:

```
Set-Item WSMAN:\localhost\service\auth\credssp –value $true
```

- Your domain policy must permit delegation of fresh credentials. In a Group Policy object (GPO), this is found in Computer Configuration > Policies > Administrative Templates > System > Credential Delegation > Allow Delegation of Fresh Credentials. You must provide the names of the machines to which credentials may be delegated, or specify a wildcard like "*.ad2008r2.loc" to allow an entire domain. Be sure to allow time for the updated GPO to apply, or run Gpupdate on the originating computer (or reboot it).

> **Note** Once again, the name you provide here is important. Whatever you'll actually be typing for the –computerName parameter is what must appear here. This makes it really tough to delegate credentials to, say, IP addresses, without just adding "*" as an allowed delegate. Adding "*," of course, means you can delegate to ANY computer, which is potentially dangerous, as it makes it easier for an attacker to impersonate a machine and get hold of your super-privileged Domain Admin account!

- When running a Remoting command, you must specify the "-Authentication CredSSP" parameter. You must also use the –Credential parameter and supply a valid

DOMAIN\Username (you'll be prompted for the password) – even if it's the same username that you used to open PowerShell in the first place.

After setting the above, we were able to use Enter-PSSession to go from our domain controller to my member server, and then use Invoke-Command to run a command on a client computer – the connection illustrated in figure 2.25.

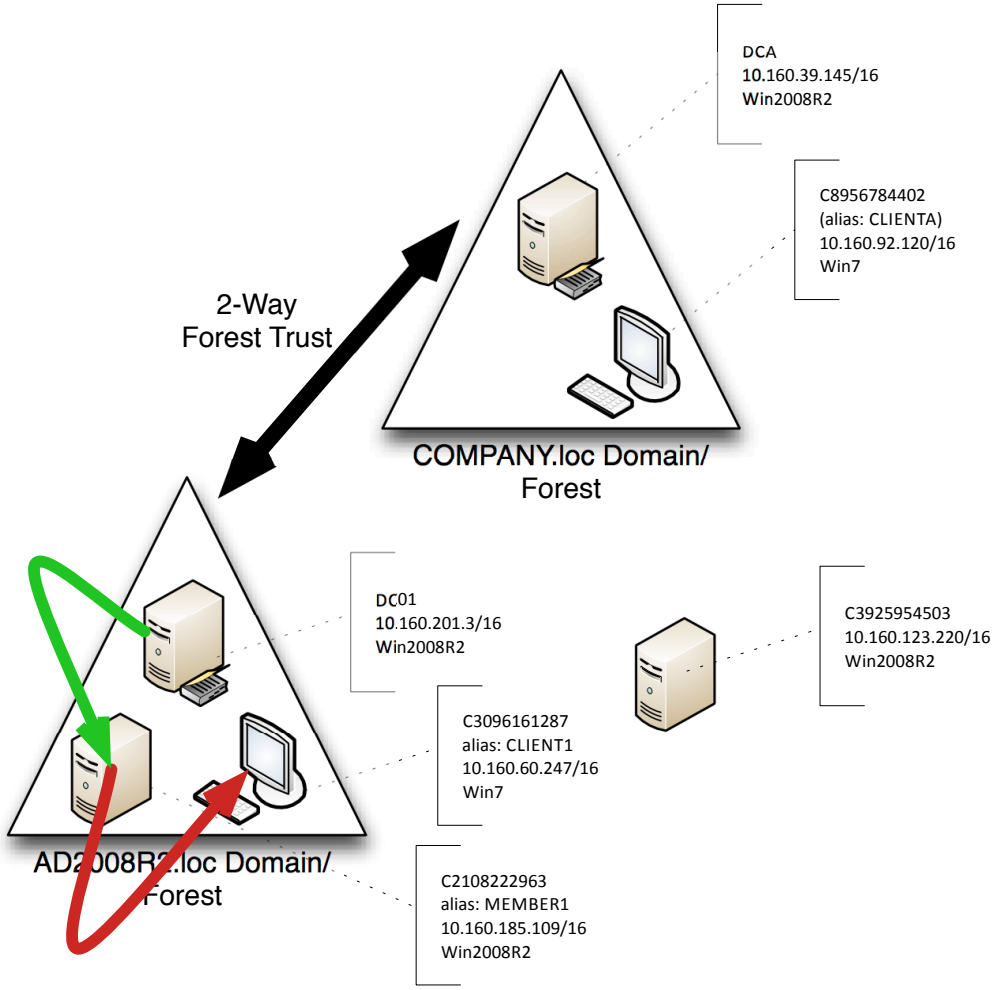


Figure 2.25 The connections for the second-hop test

Seem tedious and time-consuming to make all of those changes? There's a faster way. On the originating computer, run this:

```
Enable-WSManCredSSP –Role Client –Delegate name
```

Where “name” is the name of the computers that you plan to remote to next. This can be a wildcard, like *, or a partial wildcard, like *.AD2008R2.loc. Then, on the intermediate computer (the one to which you will delegate your credentials), run this:

```
Enable-WSManCredSSP –Role Server
```

Between them, these two commands will accomplish almost all of the configuration points we listed earlier. The only exception is that they will modify your local policy to permit fresh credential delegation, rather than modifying domain policy via a GPO. You can choose to modify the domain policy yourself, using the GPMC, to make that particular setting more universal.

Working with Endpoints

As you learned at the beginning of this guide, Remoting is designed to work with multiple different endpoints on a computer. In PowerShell terminology, each endpoint is a *session configuration*, or just a *configuration*. Each can be configured to offer specific services and capabilities, as well as having specific restrictions and limitations.

Connecting to a Different Endpoint

When you use a command like Invoke-Command or Enter-PSSession, you normally connect to a remote computer's default endpoint. That's what we've done up to now. But you can see the other enabled endpoints by running Get-PSSessionConfiguration, as shown in figure 3.1.

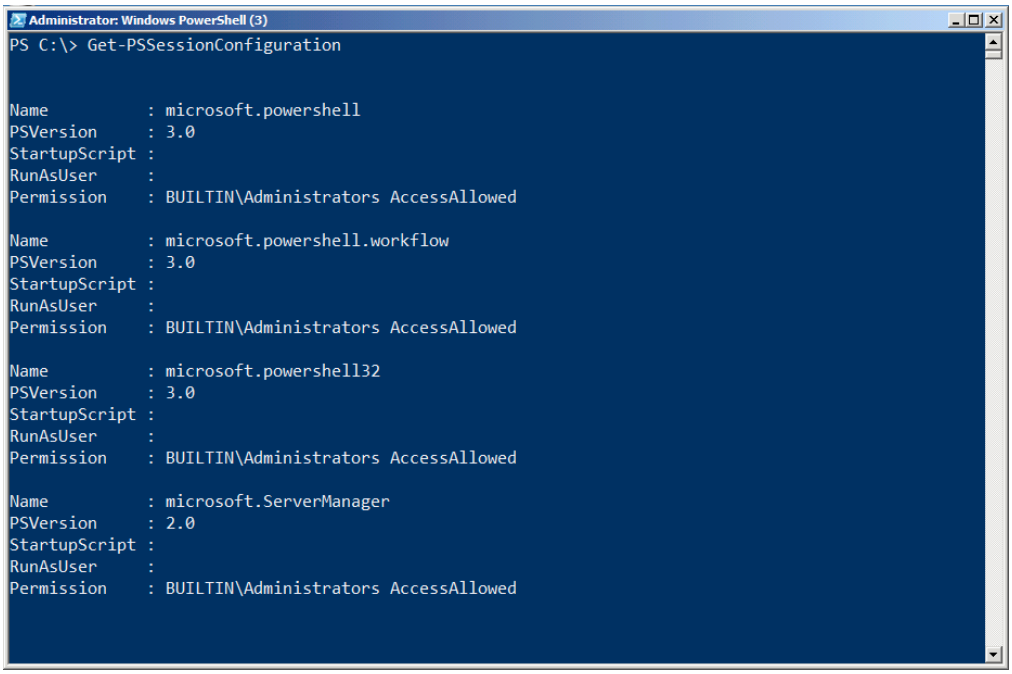


Figure 3.1 Listing the installed endpoints

Note As we pointed out in an earlier chapter, every computer will show different defaults endpoints. Our output was from a Windows Server 2008 R2

computer, which has fewer default endpoints than, say, a Windows 2012 computer.

Each endpoint has a name, such as “Microsoft.PowerShell” or “Microsoft.PowerShell32.” To connect to a specific endpoint, add the –ConfigurationName parameter to your Remoting command, as shown in Figure 3.2.

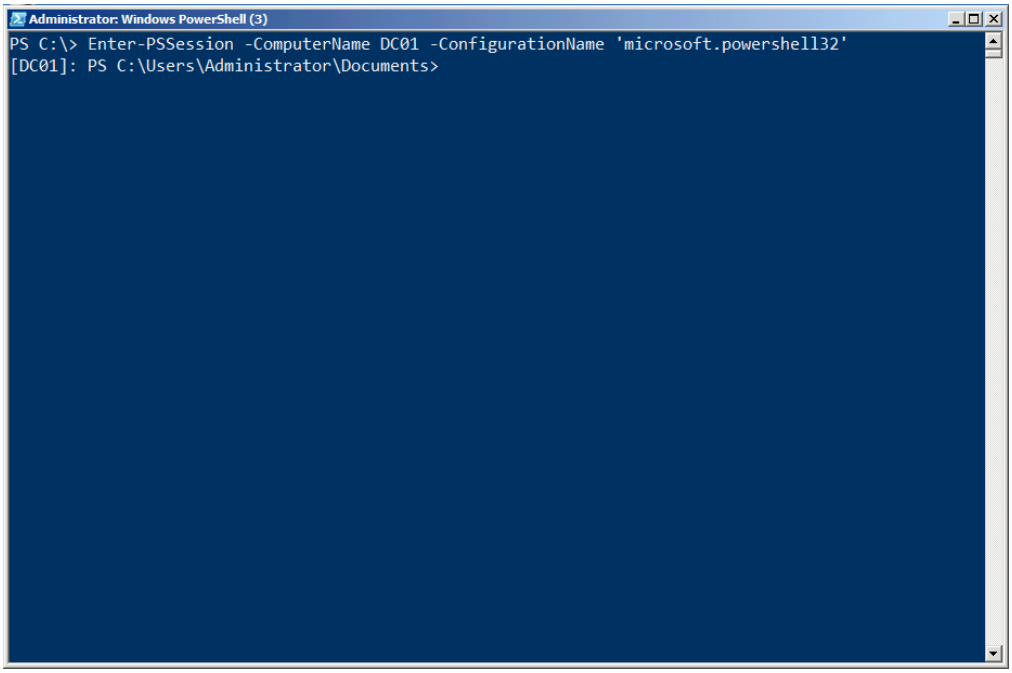


Figure 3.2 Connecting to a specific configuration (endpoint) by name

Creating a Custom Endpoint

There are a number of reasons to create a custom endpoint (or configuration):

- You can have scripts and modules auto-load whenever someone connects.
- You can specify a security descriptor (SDDL) that determines who is allowed to connect.
- You can specify an alternate account that will be used to run all commands within the endpoint – as opposed to using the credentials of the connected users.
- You can limit the commands that are available to connected users, thus restricting their capabilities.

There are two steps in setting up an endpoint: Creating a session configuration file which will define the endpoints capabilities, and then registering that file, which enables the endpoint and defines its configurations. Figure 3.3 shows the help for the New-PSSessionConfigurationFile command, which accomplishes the first of these two steps.

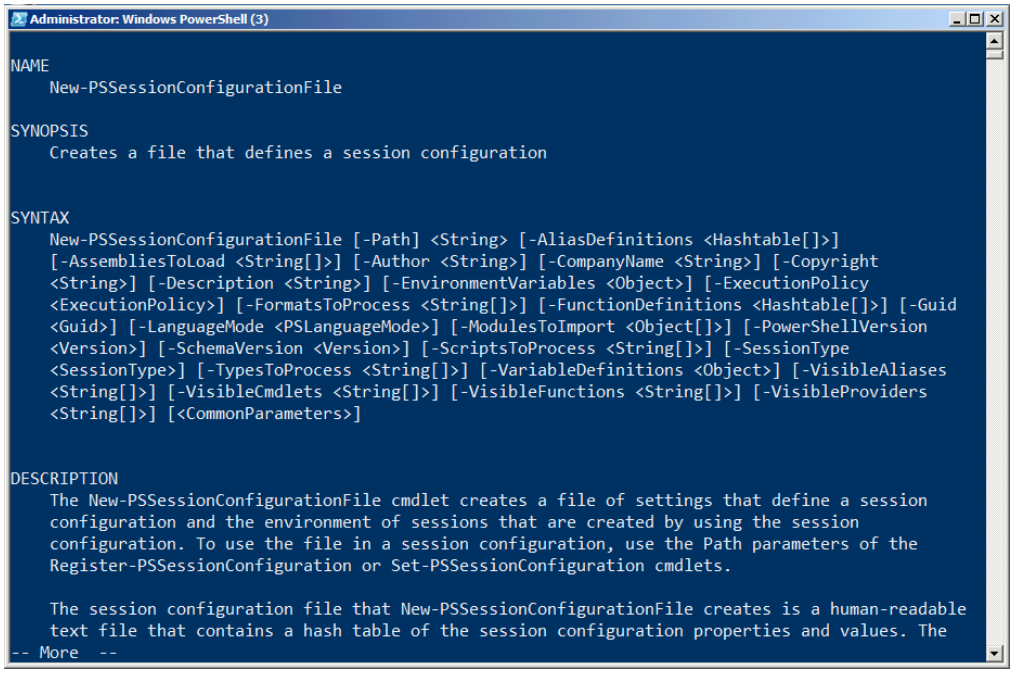


Figure 3.3 The New-PSSessionConfigurationFile command

Here's some of what the command allows you to specify (review the help file yourself for the other parameters):

- -Path: The only mandatory parameter, this is the path and filename for the configuration file you'll create. Name it whatever you like, and use a .PSSC filename extension.
- -AliasDefinitions: This is a hash table of aliases and their definitions. For example, `@{Name='d';Definition='Get-ChildItem';Options='ReadOnly'}` would define the alias d. Use a comma-separated list of these hash tables to define multiple aliases.
- -EnvironmentVariables: A single hash table of environment variables to load into the endpoint: `@{'MyVar'='\\SERVER\Share';'MyOtherVar'='SomethingElse'}`
- -ExecutionPolicy: Defaults to Restricted if you don't specify something else; use Unrestricted, AllSigned, or RemoteSigned. This sets the script execution policy for the endpoint.

- -FormatsToProcess and –TypesToProcess: Each of these is a comma-separated list of path and filenames to load. The first specifies .format.ps1xml files that contain view definitions, while the second specifies a .ps1xml file for PowerShell's Extensible Type System (ETS).
- -FunctionDefinitions: A comma-separated list of hash tables, each of which defines a function to appear within the endpoint. For example,
  `@{Name='MoreDir';Options='ReadOnly';Value={ Dir | more }}`
- -LanguageMode: The mode for PowerShell's script language. "FullLanguage" and "NoLanguage" are options; the latter permits only functions and cmdlets to run. There's also "RestrictedLanguage" which allows a very small subset of the scripting language to work – see the help for details.
- -ModulesToImport: A comma-separated list of module names to load into the endpoint. You can also use hash tables to specify specific module versions; read the command's full help for details.
- -PowerShellVersion: '2.0' or '3.0,' specifying the version of PowerShell you want the endpoint to use. 2.0 can only be specified if PowerShell v2 is independently installed on the computer hosting the endpoint (installing v3 "on top of" v2 allows v2 to continue to exist).
- -ScriptsToProcess: A comma-separated list of path and file names of scripts to run when a user connects to the endpoint. You can use this to customize the endpoint's runspace, define functions, load modules, or do anything else a script can do. However, in order to run, the script execution policy must permit the script.
- -SessionType: "Empty" loads nothing by default, leaving it up to you to load whatever you like via script or the parameters of this command. "Default" loads the normal PowerShell core extensions, plus whatever else you've specified via parameter. "RestrictedRemoteServer" adds a fixed list of seven commands, plus whatever you've specified; see the help for details on what's loaded.

Caution Some commands are important – like Exit-PSSession, which enables someone to cleanly exit an interactive Remoting session. RestrictedRemoteServer loads these, but Empty does not.

- -VisibleAliases, -VisibleCmdlets, –VisibleFunctions, and -VisibleProviders: These comma-separated lists define which of the aliases, cmdlets, functions, and PSProviders you've loaded will actually be visible to the endpoint user. These enable you to load an entire module, but then only expose one or two commands, if desired.

Note You can't use a custom endpoint alone to control which parameters a user will have access to. If you need that level of control, one option is to dive into .NET Framework programming, which does allow you to create a more fine-grained remote configuration. That's beyond the scope of this guide. You could also create a custom endpoint that only included *proxy functions*, another

way of "wrapping" built-in commands and adding or removing parameters – but that's also beyond the scope of this guide.

Once you've created the configuration file, you're ready to register it. This is done with the Register-PSSessionConfiguration command, as shown in figure 3.4.

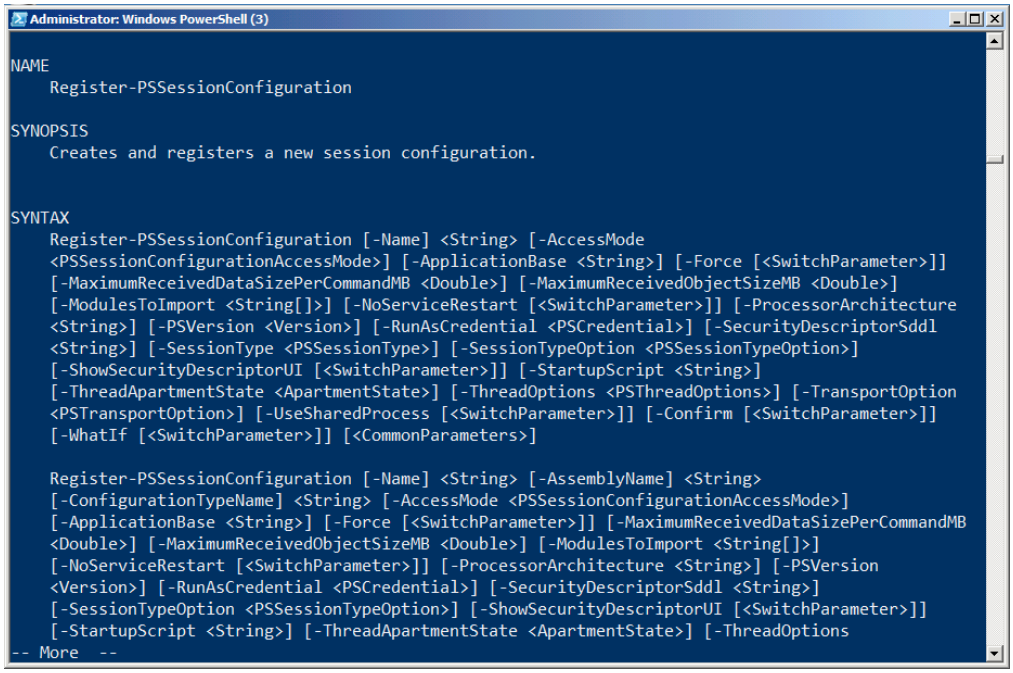


Figure 3.4 The Register-PSSessionConfiguration command

As you can see, there's a lot going on with this command. Some of the more interesting parameters include:

- -RunAsCredential: This lets you specify a credential that will be used to run all commands within the endpoint. Providing this credential enables users to connect and run commands that they normally wouldn't have permission to run; by limiting the available commands (via the session configuration file), you can restrict what users can do with this elevated privilege.
- -SecurityDescriptorSddl: This lets you specify who can connect to the endpoint. The specifier language is complex; consider using –ShowSecurityDescriptorUI instead, which shows a graphical dialog box to set the endpoint permissions.
- -StartupScript: This specifies a script to run each time the endpoint starts.

You can explore the other options on your own in the help file. Let's take a look at actually creating and using one of these custom endpoints. As shown in figure 3.5, we've created a new AD user account for SallyS of the Sales department. Sally, for some reason, needs to be able to list the users in our AD domain – but that's all she must be able to do. As-is, her account doesn't actually have permission to do so.

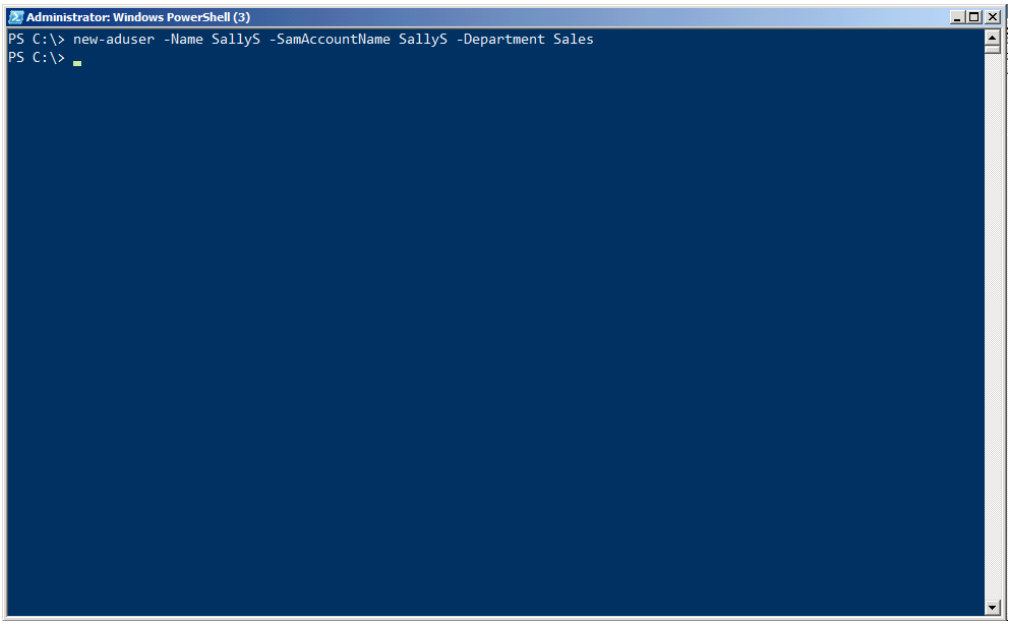
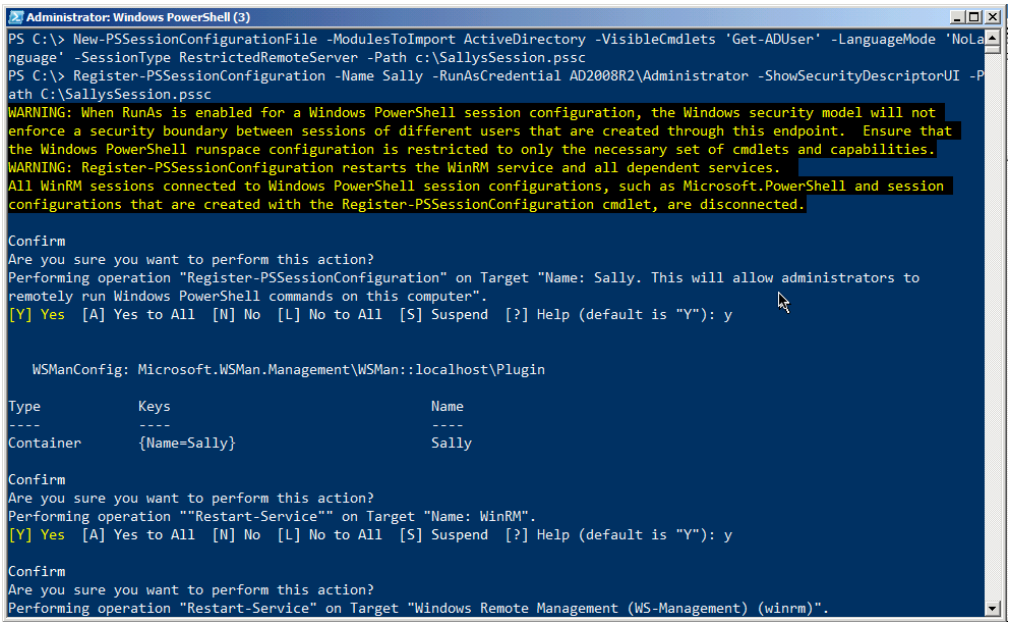
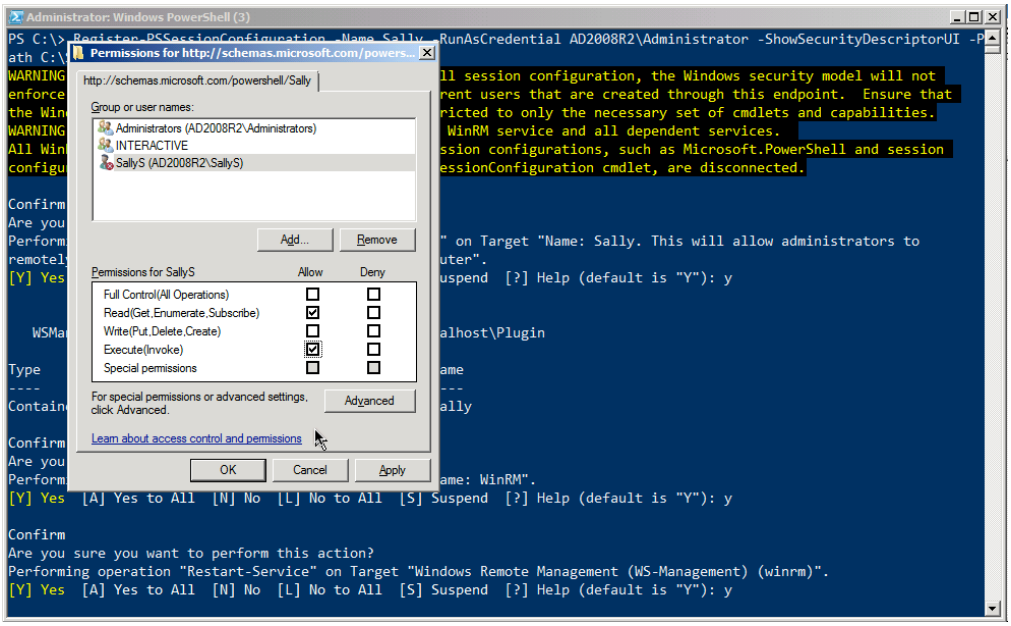


Figure 3.5 Creating a new AD user account to test

Figure 3.6 shows the creation of the new session configuration file, and the registration of the session. Notice that the session will auto-import the ActiveDirectory module, but only make the Get-ADUser cmdlet visible to Sally. We've specified a restricted remote session type, which will provide a few other key commands to Sally. We also disabled PowerShell's scripting language. When registering the configuration, we specified a "Run As" credential (we were prompted for the password), which is the account all commands will actually execute as.



Because we used the –ShowSecurityDescriptorUI, we got a dialog box like the one shown in figure 3.7. This is an easier way of setting the permissions for who can use this new endpoint. Keep in mind that the endpoint will be running commands under a Domain Admin account, so we want to be very careful who we actually let in! Sally needs, at minimum, Execute and Read permission, which we've given her.



We then set a password for Sally and enabled her user account. Everything up to this point has been done on the DC01.AD2008R2.loc computer; figure 3.8 moves to that domain's Windows 7 client computer, where we logged in using Sally's account. As you can see, she was unable to enter the default session on the domain controller. But when she attempted to enter the special new session we set up just for her, she was successful. She was able to run Get-ADUser as well.

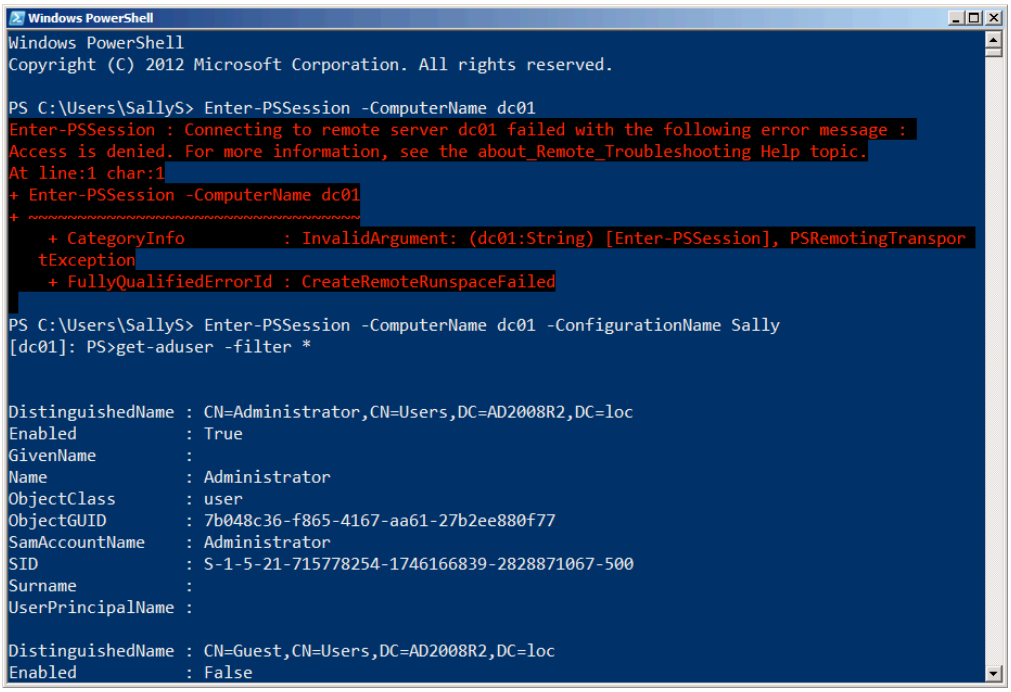
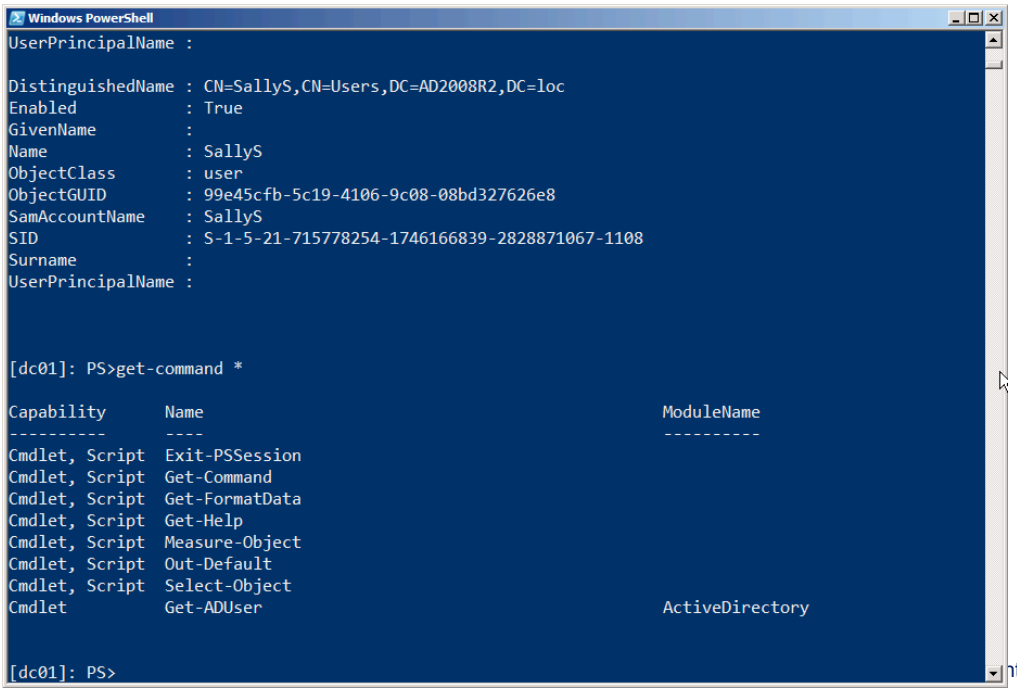


Figure 3.9 confirms that Sally has a very limited number of commands to play with. Some of these commands – like Get-Help and Exit-PSSession – are pretty crucial for using the endpoint. Others, like Select-Object, give Sally a minimal amount of non-destructive convenience for getting her command output to look like she needs. This command list (aside from Get-ADUser) is automatically set when you specify the "restricted remote" session type in the session configuration file.



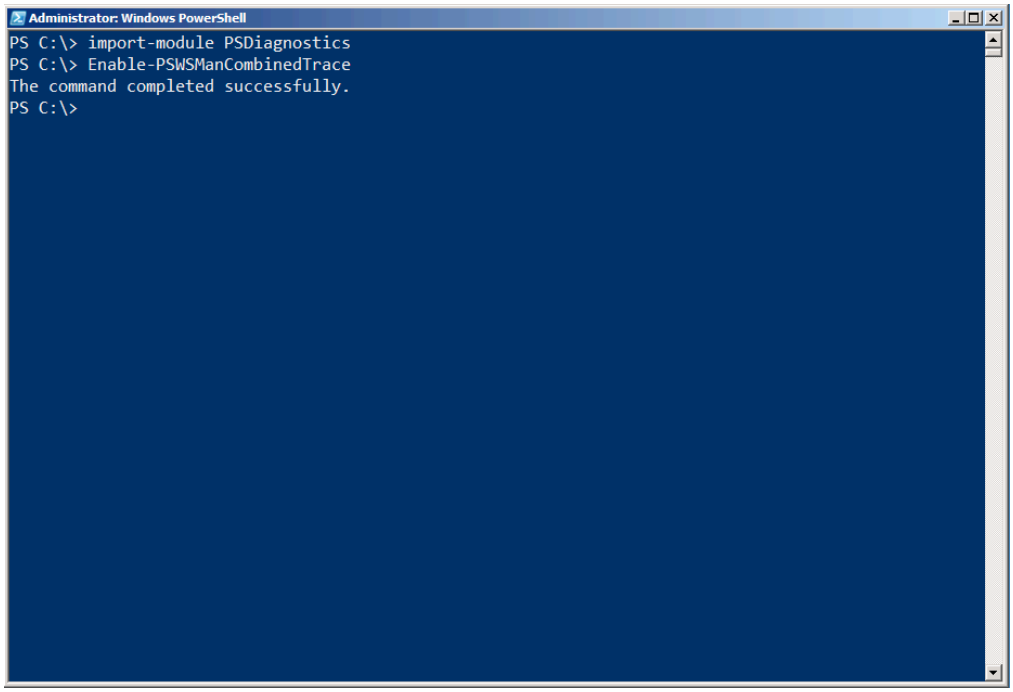
In reality, it's unlikely that a Sales user like Sally would be running commands in the PowerShell console. More likely, she'd use some GUI-based application that ran the commands "behind the scenes." Either way, we've ensured that she has exactly the functionality she needs to do her job, and nothing more.

Diagnostics and Troubleshooting

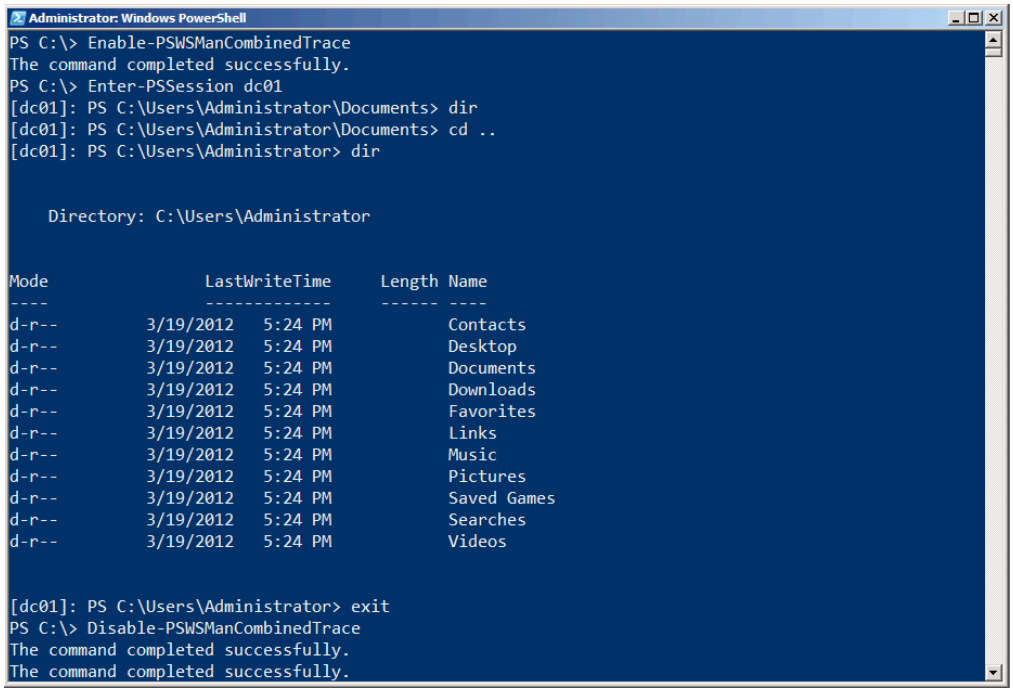
Troubleshooting and diagnosing Remoting can be one of the most difficult tasks an administrator has to deal with. When Remoting works, it works; when it doesn't, it's often hard to tell why. Fortunately, PowerShell v3 and its accompanying implementation of Remoting have much clearer and more prescriptive error messages than prior versions did. However, even v2 included an undocumented and little-appreciated module named PSDiagnostics, which is designed specifically to facilitate Remoting troubleshooting. Essentially, the module lets you turn on detailed trace log information before you attempt to initiate a Remoting connection. You can then utilize that detailed log information to get a better idea of where Remoting is failing.

Diagnostics Examples

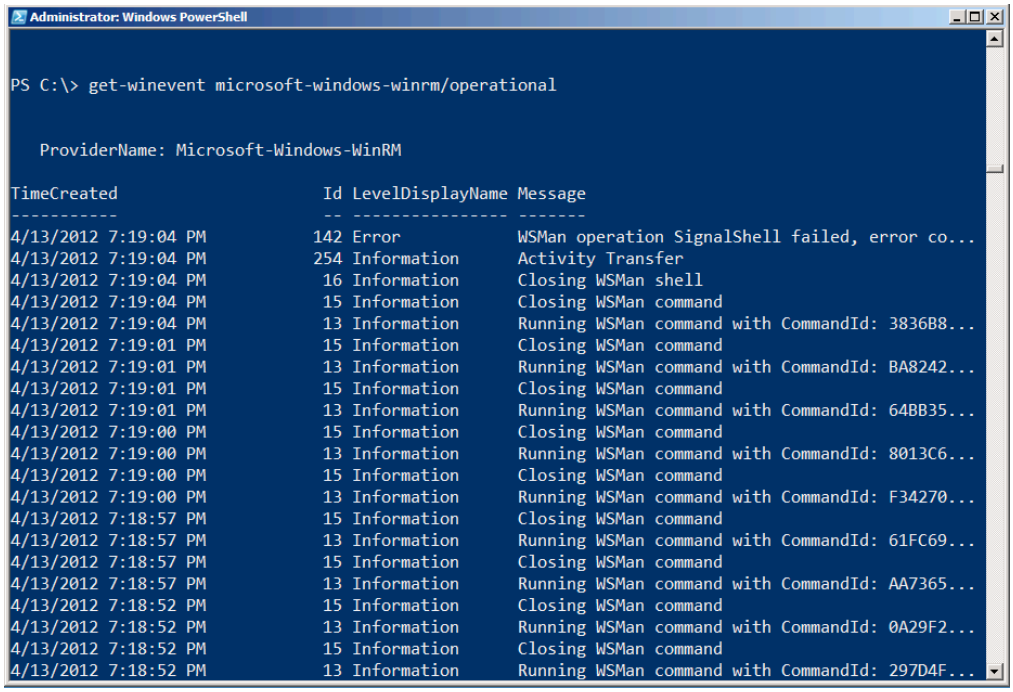
For the following scenarios, we started by importing the PSDiagnostics module (note that this is implemented as a script module, and requires an execution policy that permits it to run, such as RemoteSigned or Unrestricted). Figure 4.1 also shows that we ran the Enable-PSWSManCombinedTrace command, which starts the extended diagnostics logging.



For each scenario, we then ran one or more commands that involved Remoting, as demonstrated in figure 4.2. We then disabled the trace by running Disable-PSWSManCombinedTrace, so that the log would only contain the details from that particular attempt (we cleared the log between attempts, so that each scenario provided a fresh diagnostics log).

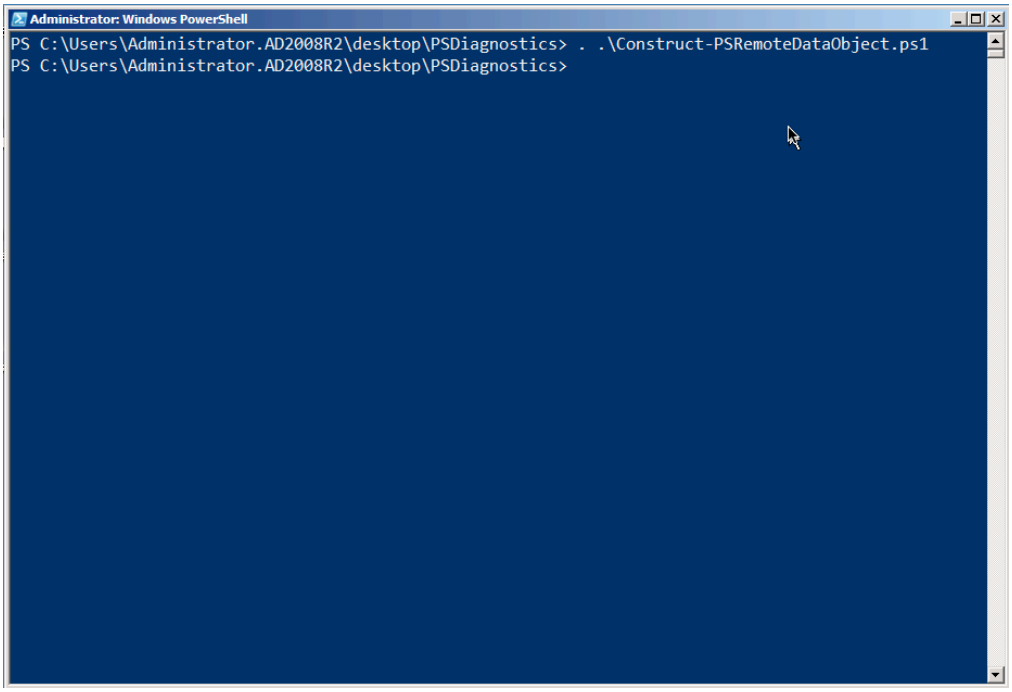


Finally, as shown in figure 4.3, we retrieved the messages from the log. In the scenarios that follow, we'll provide an annotated version of these. Note that we'll typically truncate much of this output so that we can focus on the most meaningful pieces. Also note that there's a bit of a difference in reading the information from the event log architecture, as we're doing in figure 4.3, and reading the .EVT trace file directly, as we'll do in some of our scenarios. The latter will provide combined information from different logs, which can sometimes be more useful.



We're also going to be making use of the Microsoft-Windows-WinRM/analytic log, which does not normally contain human-readable information. In order to utilize the log's contents, we'll use an internal Microsoft utility (which we've been given permission to distribute; you'll find it at http://files.concentratedtech.com/psdiagnostics.zip) to translate the log's contents into something we can read.

Trace information is stored in PowerShell's installation folder (run `cd $pshome` to get there, then change to the Traces folder). The filename extension is .ETL, and you can use `Get-WinEvent -path filename.etl` to read a particular file. The Construct-PSRemoteDataObject command, included in the ZIP file we referenced, can translate portions of the Analytic log's Message property into human-readable text. A demo script included in the ZIP file shows how to use it. As shown in figure 4.4, we dot-sourced the Construct-PSRemoteDataObject.ps1 file into our shell in order to gain access to the commands it contains.



We also deleted the contents of C:\Windows\System32\WindowsPowerShell\v1.0\Traces prior to starting each of the following examples.

A Perfect Remoting Connection

For this connection, we went from the Windows 7 client computer in the AD2008R2 domain to the DC01 domain controller. On the DC, we changed to the C:\ folder, ran a directory, and then ended the session. Figure 4.5 shows the entire scenario.

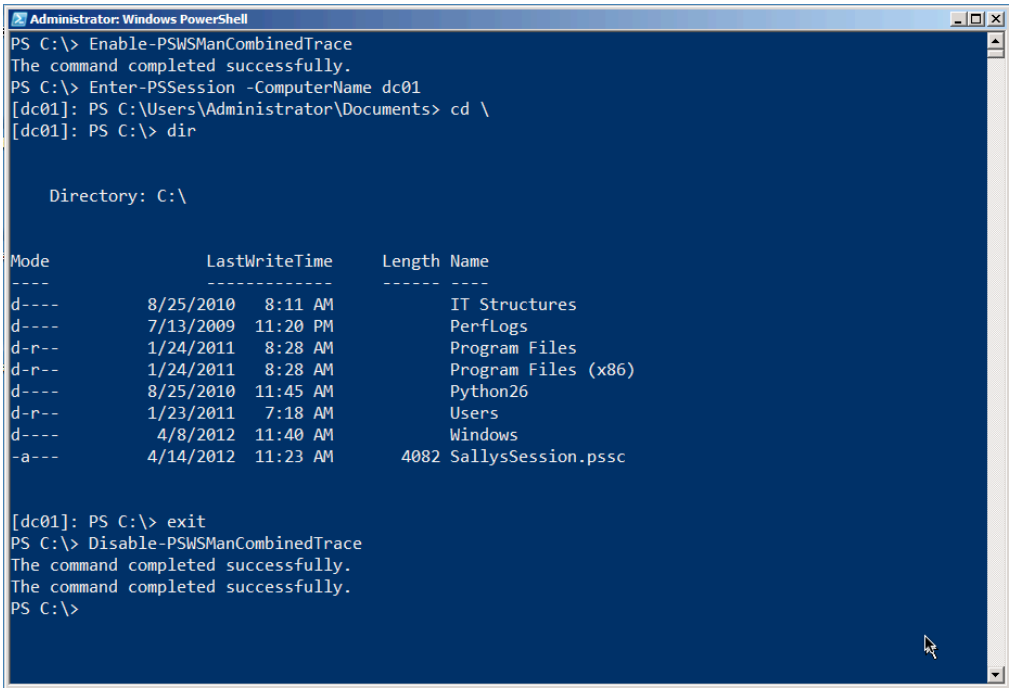


Figure 4.5 The example for this scenario

We then read the log in chronological order. You need to be a bit careful; running Enable-PSWSManCombinedTrace and Disable-PSWSManCombined trace actually create log events themselves. We'll often run the Enable command, and then wait a few minutes to actually do anything with Remoting. That way, we can tell by the timestamp in the log when the "real" traffic began. We'll wait a few more minutes before running the Disable command, again so that we can easily tell when the "real" log traffic ended. Also note that we'll be getting information from two logs, WinRM and PowerShell, although reading the .ETL file with Get-WinEvent will grab everything in sequence.

> **Note** We've experienced problems using Get-WinEvent in PowerShell v3 on non-US English machines. If you run into problems, consider running the command from PowerShell v2, or use the GUI Event Viewer application to view the event log.

The connection begins with (in this example) Enter-PSSession and name resolution, as shown in figure 4.6.

```
4/14/2012 3:03:39 PM Command Enter-PSSession is Started.

                    Context:
                            Severity = Informational
                            Host Name = ConsoleHost
                            Host Version = 3.0
                            Host ID = 5daaddbe-8c9d-4ee4-ab44-fac774eedc6f
                            Engine Version = 3.0
                            Runspace ID = f47408cf-bd95-4ced-ace8-e799421d646b
                            Pipeline ID = 294
                            Command Name = Enter-PSSession
                            Command Type = Cmdlet
                            Script Name =
                            Command Path =
                            Sequence Number = 89
                            User = AD2008R2\Administrator
                            Shell ID = Microsoft.PowerShell

                    User Data:

4/14/2012 3:03:39 PM ComputerName resolved to localhost
4/14/2012 3:03:39 PM ComputerName resolved to dc01
4/14/2012 3:03:39 PM ComputerName resolved to dc01
4/14/2012 3:03:39 PM ComputerName resolved to dc01
```

Figure 4.6 Starting the Remoting connection

WinRM has to spin up a runspace (essentially, a PowerShell process) on the remote computer. That includes setting several options for locale, timing, and so on, as shown in figure 4.7.

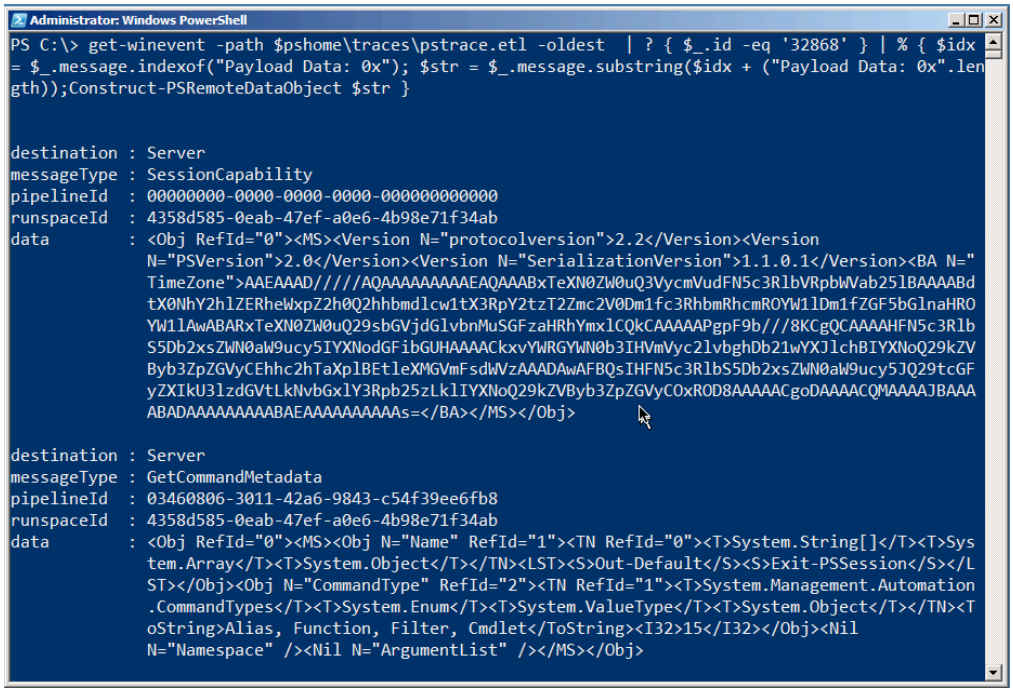
```
4/14/2012 3:03:39 PM Creating Runspace object
                        Instance Id: cd32125b-0290-4887-89a9-910ca224b3f7
4/14/2012 3:03:39 PM Creating RunspacePool object
                        InstanceId 4358d585-0eab-47ef-a0e6-4b98e71f34ab
                        MinRunspaces 1
                        MaxRunspaces 1
4/14/2012 3:03:39 PM Creating WSMan Session. The connection string is: dc01/wsman?PSVersion=3.0
4/14/2012 3:03:39 PM WSMan Create Session operation completed successfuly
4/14/2012 3:03:39 PM Getting WSMan Session Option (29) - INVALID_SESSION_OPTION.
4/14/2012 3:03:39 PM Getting WSMan Session Option (11) - WSMAN_OPTION_MAX_RETRY_TIME.
4/14/2012 3:03:39 PM Setting WSMan Session Option (26) - WSMAN_OPTION_UI_LANGUAGE with value
                    (en-US) completed successfully.
4/14/2012 3:03:39 PM Setting WSMan Session Option (25) - WSMAN_OPTION_LOCALE with value (en-US)
                    completed successfully.
4/14/2012 3:03:39 PM Setting WSMan Session Option (1) - WSMAN_OPTION_DEFAULT_OPERATION_TIMEOUTMS
                    with value (180000) completed successfully.
4/14/2012 3:03:39 PM Setting WSMan Session Option (12) - WSMAN_OPTION_TIMEOUTMS_CREATE_SHELL with
                    value (180000) completed successfully.
4/14/2012 3:03:39 PM Setting WSMan Session Option (17) - WSMAN_OPTION_TIMEOUTMS_CLOSE_SHELL with
                    value (60000) completed successfully.
4/14/2012 3:03:39 PM Setting WSMan Session Option (16) - WSMAN_OPTION_TIMEOUTMS_SIGNAL_SHELL with
                    value (60000) completed successfully.
4/14/2012 3:03:39 PM Opening RunspacePool
4/14/2012 3:03:39 PM Runspace state changed to Opening
```

Figure 4.7 Starting the remote runspace

This will go on for a while. Eventually, you'll see WinRM beginning to send "chunks," which are packetized communications. These are sent via the Simple Object Access Protocol, so expect to see "SOAP" referenced a lot (WS-MAN is a Web service, remember, and SOAP is the communications language of Web services). Figure 4.8 shows a couple of these 1500-byte chunks. Notice that the actual payload is pretty much gibberish.

```
4/14/2012 3:03:39 PM SOAP [client sending index 1 of 6 total chunks (1500 bytes)] <s:Envelope
                    xmlns:s="http://www.w3.org/2003/05/soap-envelope"
                    xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing"
                    xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:p="http://schem
                    as.microsoft.com/wbem/wsman/1/wsman.xsd"><s:Header><a:To>http://dc01:5985/wsma
                    n?PSVersion=3.0</a:To><w:ResourceURI s:mustUnderstand="true">http://schemas.mi
                    crosoft.com/powershell/Microsoft.PowerShell</w:ResourceURI><a:ReplyTo><a:Addre
                    ss s:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/08/addressing/ro
                    le/anonymous</a:Address></a:ReplyTo><a:Action s:mustUnderstand="true">http://s
                    chemas.xmlsoap.org/ws/2004/09/transfer/Create</a:Action><w:MaxEnvelopeSize s:m
                    ustUnderstand="true">512000</w:MaxEnvelopeSize><a:MessageID>uuid:86AACA6B-4F66
                    -42A9-A267-8C287AA011E1</a:MessageID><w:Locale xml:lang="en-US"
                    s:mustUnderstand="false" /><p:DataLocale xml:lang="en-US"
                    s:mustUnderstand="false" /><p:ActivityId s:mustUnderstand="false">01911C40-F80
                    0-0000-66BA-A1FC9D15CD01</p:ActivityId><p:SessionId s:mustUnderstand="false">u
                    uid:5EB0B40D-C79B-4114-908E-8AAC76B78C42</p:SessionId><p:OperationID s:mustUnd
                    erstand="false">uuid:8076355C-7892-4C0A-9F7C-2198B60CDAF2</p:OperationID><p:Se
                    quenceId s:mustUnderstand="false">1</p:SequenceId><w:OptionSet
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    s:mustUnderstand="true"><w:Option Name="protocolversion" MustComply="true">2.2
                    </w:Option></w:OptionSet><w:OperationTimeout>PT180.000S</w:OperationTimeout><r
                    sp:CompressionType s:mustUnderstand="true" xmlns
4/14/2012 3:03:39 PM SOAP [client sending index 2 of 6 total chunks (1500 bytes)] :rsp="http://sche
                    mas.microsoft.com/wbem/wsman/1/windows/shell">xpress</rsp:CompressionType></s:
                    Header><s:Body><rsp:Shell
                    xmlns:rsp="http://schemas.microsoft.com/wbem/wsman/1/windows/shell"
                    Name="Session10"
                    ShellId="4358D585-0EAB-47EF-A0E6-4B98E71F34AB"><rsp:InputStreams>stdin pr</rsp
                    :InputStreams><rsp:OutputStreams>stdout</rsp:OutputStreams><creationXml xmlns=
                    "http://schemas.microsoft.com/powershell">AAAAAAAAACgAAAAAAAAAAAMAAALwAgAAAAIA
                    AQCF1VhDqw7vR6DmS5jnHzSrAAAAAAAAAAAAAAAAAAAAAAAAO+7vzxPYmogUmVmSWQ9IjAiPjxNUz48Vm
                    Vyc2lvbiBOPSJwcm90b2NvbHZlcnNpb24iPjIuMjwvVmVyc2lvbj48VmVyc2lvbiBOPSJQU1ZlcnNp
                    b24iPjIuMDwvVmVyc2lvbj48VmVyc2lvbiBOPSJTZXJpYWxpemF0aW9uVmVyc2lvbiI+MS4xLjAuMT
                    wvVmVyc2lvbj48QkEgTj0iVGltZVpvbmUiPkFBRUFBQUQvLy8vL0FRQUFBQUFBQUFBRUFRQUFBQnhU
                    ZVhOMFpXMHVRM1Z5Y21WdWRGTjVjM1JsYlZScGJXVmFiMjVsQkFBQUFCZHRYME5oWTJobFpFUmhlV3
                    hwWjJoMFEyaGhibWRsY3cxdFgzUnBZMnR6VDJabWMyVjBEbTFmYzNSaGJtUmhjbVJPWVcxbERtMWZa
                    R0Y1YkdsbmFIUk9ZVzFsQXdBQkFSeFRlWE4wWlcwdVEyOXNiR1ZqZEdsdmJuTXVTR0Z6YUhSaFlteG
                    xDUWtDQUFBQUFQZ3BGOWIvLy84S0NnUUNBQUFBSEZONWMzUmxiUzVEYjJ4c1pXTjBhVzl1Y3k1SVlY
                    Tm9kR0ZpYkdVSEFBQUFDa3h2WVdSR1lXTjBiM0lIVm1WeWMybHZiZ2hEYjIxd1lYSmxjaEJJWVhOb1
                    EyOWtaVkJ5YjNacFpHVnlDRWhoYzJoVGFYcGxCRXRsZVhNR1ZtRnNkV1Z6QUFBREF3QUZCUXNJSEZO
                    NWMzUmxiUzVEYjJ4c1pXTjBhVzl1Y3k1SlEyOXRjR0Z5WlhJa1UzbHpkR1Z0TGtOdmJHeGxZM1JwYj
                    I1ekxrbElZWE5vUTI5a1pWQnliM1pwWkdWeUNPeFJPRDhBQUFBQUNnb0RBQUFBQ1FNQUFBQUpCQUFB
                    QUJBREFBQUFBQUFBQUJBRUFBQUFBQUFBQUFzPTwvQkE+PC9NUz48L09iaj4AAAAAAAAAKQAAAAAAAA
                    AAAwAADfkCAAAABAABAIXVWEOrDu9HoOZLmOcfNKsAAAAA
```

This gibberish is what the Construct-PSRemoteDataObject command can translate. For example, those “sending” messages have an event ID of 32868; by looking for just those events we can see what’s being sent, as shown in figure 4.9.



In this case, the client was asking the server (which is listed as the destination) about its capabilities, and for some metadata on the Exit-PSSession command (that's the second message). This is how the client figures out what kind of server its talking to, and other important, preliminary information. Now, the client knows what version of the serialization protocol will be used to send data back and forth, what time zone the server is in, and other details.

> **Note** Event ID 32868 is client-to-server traffic; ID 32867 represents server-to-client traffic. Using those two IDs along with Construct-PSRemoteDataObject can reveal the majority of the session transcript once the connection is established.

Moving on. As shown in figure 4.10, you'll then see some authentication back-and-forth, during which some errors can be expected. The system will eventually get over it and, as shown, start receiving chunks of data from the server.

```
4/14/2012 3:03:39 PM An error was encountered while processing an operation.
                     Error Code: 11001
4/14/2012 3:03:39 PM The chosen authentication mechanism is Kerberos
4/14/2012 3:03:39 PM Sending the request for operation CreateShell to destination machine and port
                     dc01:5985
4/14/2012 3:03:39 PM An error was encountered while processing an operation.
                     Error Code: 11001
4/14/2012 3:03:39 PM The chosen authentication mechanism is Kerberos
4/14/2012 3:03:39 PM Received the response from Network layer; status: 200 (HTTP_STATUS_OK)
4/14/2012 3:03:39 PM Received the response from Network layer; status: 200 (HTTP_STATUS_OK)
4/14/2012 3:03:39 PM Activity Transfer
4/14/2012 3:03:39 PM Activity Transfer
4/14/2012 3:03:39 PM SOAP [client receiving index 1 of 2 total chunks (3000 bytes)] <s:Envelope
                     xml:lang="en-US" xmlns:s="http://www.w3.org/2003/05/soap-envelope"
                     xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing"
                     xmlns:x="http://schemas.xmlsoap.org/ws/2004/09/transfer"
                     xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"
                     xmlns:rsp="http://schemas.microsoft.com/wbem/wsman/1/windows/shell" xmlns:p="h
                     ttp://schemas.microsoft.com/wbem/wsman/1/wsman.xsd"><s:Header><a:Action>http:/
                     /schemas.xmlsoap.org/ws/2004/09/transfer/CreateResponse</a:Action><a:MessageID
                     >uuid:67E26C83-FCD7-41EA-9B26-636BBE961791</a:MessageID><p:OperationID s:mustU
                     nderstand="false">uuid:8076355C-7892-4C0A-9F7C-2198B60CDAF2</p:OperationID><p:
```

A rather surprising amount of back-and-forth can ensue as the two computers exchange pleasantries, share information about each other and how they work, and so on. We're going to switch our event log output, now, to include event ID numbers, because those can be pretty useful when trying to grab specific pieces of data. At this point, the log will mainly consist of the client sending commands and the server sending back the results. This is more readable when you use Construct-PSRemoteDataObject, so here's the complete back-and-forth from that perspective: First up is the client's statement of its session capabilities:

```
   destination : Server

   messageType : SessionCapability

   pipelineId  : 00000000-0000-0000-0000-000000000000

   runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

   data        : <Obj RefId="0"><MS><Version

                 N="protocolversion">2.2</Version><Version

                 N="PSVersion">2.0</Version><Version

                 N="SerializationVersion">1.1.0.1</Version><BA N="TimeZon

                 e">AAEAAAD/////AQAAAAAAAAAEAQAAABxTeXN0ZW0uQ3VycmVudFN5c

                 3RlbVRpbWVab25lBAAAABdtX0NhY2hlZERheWxpZ2h0Q2hhbmdlcw1tX

                 3RpY2tzT2Zmc2V0Dm1fc3RhbmRhcmROYW1lDm1fZGF5bGlnaHROYW1lA
```

```
wABARxTeXN0ZW0uQ29sbGVjdGlvbnMuSGFzaHRhYmxlCQkCAAAAAPgpF
9b///8KCgQCAAAAHFN5c3RlbS5Db2xsZWN0aW9ucy5IYXNodGFibGUHA
AAACkxvYWRGYWN0b3IHVmVyc2lvbghDb21wYXJlchBIYXNoQ29kZVByb
3ZpZGVyCEhhc2hTaXplBEtleXMGVmFsdWVzAAADAwAFBQsIHFN5c3Rlb
S5Db2xsZWN0aW9ucy5JQ29tcGFyZXIkU3lzdGVtLkNvbGxlY3Rpb25zL
klIYXNoQ29kZVByb3ZpZGVyCOxROD8AAAAACgoDAAAACQMAAAAJBAAAA
BADAAAAAAAAABAEAAAAAAAAAs=</BA></MS></Obj>
```

Then the server's:

```
destination : Client
messageType : SessionCapability
pipelineId  : 00000000-0000-0000-0000-000000000000
runspaceId  : 00000000-0000-0000-0000-000000000000
data        : <Obj RefId="0"><MS><Version
              N="protocolversion">2.2</Version><Version
              N="PSVersion">2.0</Version><Version
              N="SerializationVersion">1.1.0.1</Version></MS></Obj>
```

Next is the server's $PSVersionTable object, which lists various versioning information:

```
destination : Client
messageType : ApplicationPrivateData
```

```
pipelineId  : 00000000-0000-0000-0000-000000000000
runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab
data        : <Obj RefId="0"><MS><Obj N="ApplicationPrivateData"
              RefId="1"><TN RefId="0"><T>System.Management.Automation.
              PSPrimitiveDictionary</T><T>System.Collections.Hashtable
              </T><T>System.Object</T></TN><DCT><En><S
              N="Key">PSVersionTable</S><Obj N="Value"
              RefId="2"><TNRef RefId="0" /><DCT><En><S
              N="Key">PSVersion</S><Version
              N="Value">2.0</Version></En><En><S
              N="Key">PSCompatibleVersions</S><Obj N="Value"
              RefId="3"><TN RefId="1"><T>System.Version[]</T><T>System
              .Array</T><T>System.Object</T></TN><LST><Version>1.0</Ve
              rsion><Version>2.0</Version><Version>3.0</Version></LST>
              </Obj></En><En><S N="Key">BuildVersion</S><Version
              N="Value">6.2.8314.0</Version></En><En><S
              N="Key">PSRemotingProtocolVersion</S><Version
              N="Value">2.2</Version></En><En><S
              N="Key">WSManStackVersion</S><Version
              N="Value">3.0</Version></En><En><S
              N="Key">CLRVersion</S><Version
              N="Value">4.0.30319.261</Version></En><En><S
              N="Key">SerializationVersion</S><Version N="Value">1.1.0
              .1</Version></En></DCT></Obj></En></DCT></Obj></MS></Obj
              >
```

Next the server sends information about the runspace that will be used:

```
destination : Client

messageType : RunspacePoolStateInfo

pipelineId  : 00000000-0000-0000-0000-000000000000

runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

data        : <Obj RefId="0"><MS><I32

              N="RunspaceState">2</I32></MS></Obj>
```

The client sends information about its Exit-PSSession command:

```
destination : Server

messageType : GetCommandMetadata

pipelineId  : 03460806-3011-42a6-9843-c54f39ee6fb8

runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

data        : <Obj RefId="0"><MS><Obj N="Name" RefId="1"><TN RefId="0"

              ><T>System.String[]</T><T>System.Array</T><T>System.Obje

              ct</T></TN><LST><S>Out-Default</S><S>Exit-PSSession</S><

              /LST></Obj><Obj N="CommandType" RefId="2"><TN RefId="1">

              <T>System.Management.Automation.CommandTypes</T><T>Syste

              m.Enum</T><T>System.ValueType</T><T>System.Object</T></T

              N><ToString>Alias, Function, Filter,
```

```
Cmdlet</ToString><I32>15</I32></Obj><Nil N="Namespace"

/><Nil N="ArgumentList" /></MS></Obj>
```

A bit later we'll see the result of the CD C:\ command, which is the new PowerShell prompt reflecting the new folder location:

```
destination : Client

messageType : PowerShellOutput

pipelineId  : c913b8ae-2802-4454-9d9b-926ca6032018

runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

data        : <S>PS C:\&gt; </S>
```

Next we'll look at the output of the Dir command. This first bit is writing the column headers for Mode, LastWriteTime, Length, Name, and so forth. This is all being sent to our client – we'll just include the first few lines, each of which comes across in its own block:

```
destination : Client

messageType : RemoteHostCallUsingPowerShellHost

pipelineId  : c259c891-516a-46a7-b287-27c96ff86d5b

runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

data        : <Obj RefId="0"><MS><I64 N="ci">-100</I64><Obj N="mi"

              RefId="1"><TN RefId="0"><T>System.Management.Automation.

              Remoting.RemoteHostMethodId</T><T>System.Enum</T><T>Syst

              em.ValueType</T><T>System.Object</T></TN><ToString>Write
```

```
                Line2</ToString><I32>16</I32></Obj><Obj N="mp"
                RefId="2"><TN RefId="1"><T>System.Collections.ArrayList<
                /T><T>System.Object</T></TN><LST><S>Mode
                LastWriteTime     Length Name
                                       </S></LST></Obj></MS></Obj>



destination : Client
messageType : RemoteHostCallUsingPowerShellHost
pipelineId  : c259c891-516a-46a7-b287-27c96ff86d5b
runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab
data        : <Obj RefId="0"><MS><I64 N="ci">-100</I64><Obj N="mi"
                RefId="1"><TN RefId="0"><T>System.Management.Automation.
                Remoting.RemoteHostMethodId</T><T>System.Enum</T><T>Syst
                em.ValueType</T><T>System.Object</T></TN><ToString>Write
                Line2</ToString><I32>16</I32></Obj><Obj N="mp"
                RefId="2"><TN RefId="1"><T>System.Collections.ArrayList<
                /T><T>System.Object</T></TN><LST><S>----
                -------------     ------ ----
                                       </S></LST></Obj></MS></Obj>



destination : Client
```

```
messageType : RemoteHostCallUsingPowerShellHost
pipelineId  : c259c891-516a-46a7-b287-27c96ff86d5b
runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab
data        : <Obj RefId="0"><MS><I64 N="ci">-100</I64><Obj N="mi"
              RefId="1"><TN RefId="0"><T>System.Management.Automation.
              Remoting.RemoteHostMethodId</T><T>System.Enum</T><T>Syst
              em.ValueType</T><T>System.Object</T></TN><ToString>Write
              Line2</ToString><I32>16</I32></Obj><Obj N="mp"
              RefId="2"><TN RefId="1"><T>System.Collections.ArrayList<
              /T><T>System.Object</T></TN><LST><S>d----
              8/25/2010   8:11 AM            IT Structures
                                           </S></LST></Obj></MS></Obj>



destination : Client
messageType : RemoteHostCallUsingPowerShellHost
pipelineId  : c259c891-516a-46a7-b287-27c96ff86d5b
runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab
data        : <Obj RefId="0"><MS><I64 N="ci">-100</I64><Obj N="mi"
              RefId="1"><TN RefId="0"><T>System.Management.Automation.
              Remoting.RemoteHostMethodId</T><T>System.Enum</T><T>Syst
              em.ValueType</T><T>System.Object</T></TN><ToString>Write
              Line2</ToString><I32>16</I32></Obj><Obj N="mp"
              RefId="2"><TN RefId="1"><T>System.Collections.ArrayList<
```

```
/T><T>System.Object</T></TN><LST><S>d----

7/13/2009  11:20 PM            PerfLogs

                                </S></LST></Obj></MS></Obj>
```

Eventually the command finishes and we get the prompt again:

```
destination : Client

messageType : PowerShellOutput

pipelineId  : f5c8bc7a-ec54-4180-b2d4-86479f9ea4b9

runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

data        : <S>PS C:\&gt; </S>
```

You'll also see periodic exchanges about the state of the pipeline – this indicates that the command is done:

```
destination : Client

messageType : PowerShellStateInfo

pipelineId  : f5c8bc7a-ec54-4180-b2d4-86479f9ea4b9

runspaceId  : 4358d585-0eab-47ef-a0e6-4b98e71f34ab

data        : <Obj RefId="0"><MS><I32

              N="PipelineState">4</I32></MS></Obj>
```

There's definitely a lot of data passing back and forth – but it's possible to make sense of it using these tools. Frankly, most Remoting problems take place during the connection phase, meaning once that's completed successfully you have no further problems. So in the next scenarios, we'll focus on specific connection errors.

> **Note** To clear the log and prepare for a new trace, we usually delete the .ETL files and go into Event Viewer to clear the Applications and Services Logs > Microsoft > Windows > Windows Remote Management log. If you're getting errors when running Enable-PSWSManCombinedTrace, one of those two tasks probably hasn't been completed.

Connection Problem: Blocked Port

Figure 4.11 shows what happens when you try to connect to a computer and the necessary port – 5985 by default – isn't open all the way through. We're going to look at how this appears in the log. Note that we're assuming you've already checked the computer name, made sure it resolves to the proper IP address, and so forth; what you're looking at is *definitely* a blocked port (because we set it up that way) in this example.

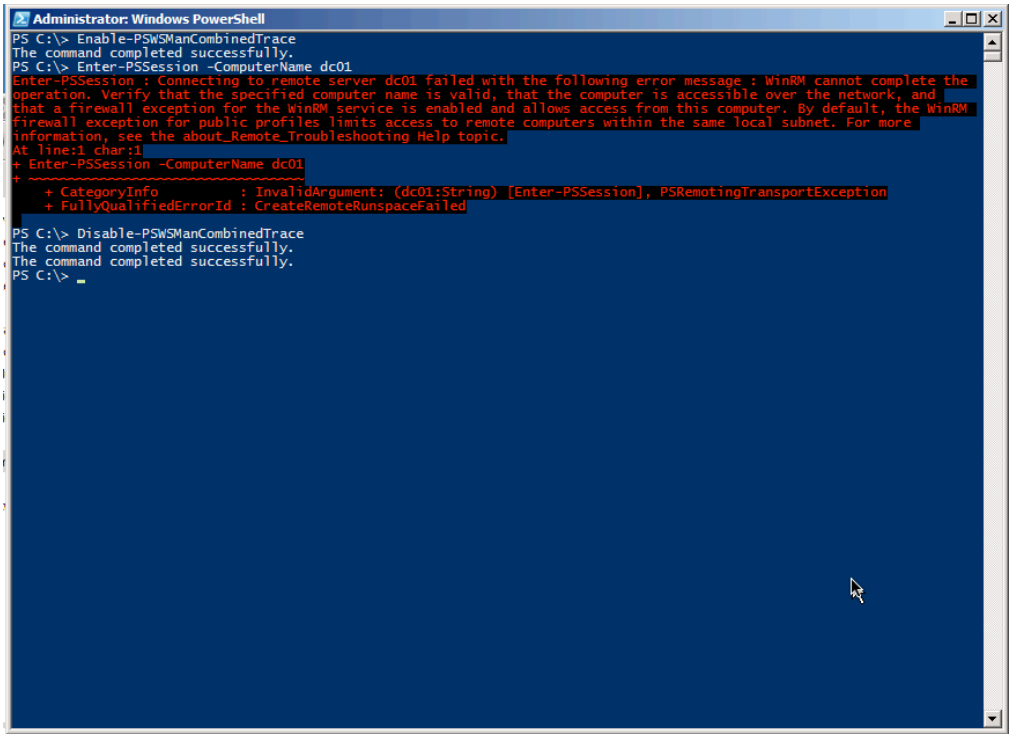


Figure 4.12 shows that we successfully resolved the computer name. We find that testing with Enter-PSSession is easiest, because it's really easy to spot that command in the log and see when the "real" log data begins.

```
7937 4/14/2012 4:01:11 PM Command Enter-PSSession is Started.

                          Context:
                                  Severity = Informational
                                  Host Name = ConsoleHost
                                  Host Version = 3.0
                                  Host ID =
                          8fd53e17-bd16-456a-8c6e-174acb89ce8c
                                  Engine Version = 3.0
                                  Runspace ID =
                          f42d6bb0-43d6-4f7d-81e9-376113a63428
                                  Pipeline ID = 66
                                  Command Name = Enter-PSSession
                                  Command Type = Cmdlet
                                  Script Name =
                                  Command Path =
                                  Sequence Number = 255
                                  User = AD2008R2\Administrator
                                  Shell ID = Microsoft.PowerShell


                          User Data:

12035 4/14/2012 4:01:11 PM ComputerName resolved to localhost
12035 4/14/2012 4:01:11 PM ComputerName resolved to dc01
12035 4/14/2012 4:01:11 PM ComputerName resolved to dc01
12035 4/14/2012 4:01:11 PM ComputerName resolved to dc01
```

Note that a lot of the initial log traffic is still WinRM talking to itself, getting set up for the actual connection attempt. Just keep scrolling through that until you start to see problem indications. Figure 4.13 shows a timeout – never a good sign – and the error message generated by WinRM. As you can see, this is exactly what we got on-screen, so PowerShell isn't hiding anything from us.

```
 138 4/14/2012 4:01:34 PM The client got a timeout from the network layer
                          (ERROR_WINHTTP_TIMEOUT)
1840 4/14/2012 4:01:34 PM An error was encountered while processing an
                          operation.
                          Error Code: 2150859046
                          Error String:<f:WSManFault xmlns:f="http://schemas.mi
                          crosoft.com/wbem/wsman/1/wsmanfault"
                          Code="2150859046"
                          Machine="C3096161287.AD2008R2.loc"><f:Message>WinRM
                          cannot complete the operation. Verify that the
                          specified computer name is valid, that the computer
                          is accessible over the network, and that a firewall
                          exception for the WinRM service is enabled and
                          allows access from this computer. By default, the
                          WinRM firewall exception for public profiles limits
                          access to remote computers within the same local
                          subnet. </f:Message></f:WSManFault>
1840 4/14/2012 4:01:34 PM An error was encountered while processing an
                          operation.
                          Error Code: 2150859046
                          Error String:<f:WSManFault xmlns:f="http://schemas.mi
                          crosoft.com/wbem/wsman/1/wsmanfault"
                          Code="2150859046"
                          Machine="C3096161287.AD2008R2.loc"><f:Message>WinRM
                          cannot complete the operation. Verify that the
                          specified computer name is valid, that the computer
                          is accessible over the network, and that a firewall
                          exception for the WinRM service is enabled and
                          allows access from this computer. By default, the
                          WinRM firewall exception for public profiles limits
                          access to remote computers within the same local
                          subnet. </f:Message></f:WSManFault>
```

This is actually one of the trickiest bits of Remoting: It can't tell why the server didn't respond. It doesn't *realize* that the port isn't open. For all WinRM knows, we could have specified a computer name that doesn't exist. All it knows is that it sent a message out to the network, and nobody replied. In the end, nearly all of the possible "low level" problems – bad IP address, bad computer name, blocked port, and so forth all look the same from WinRM's point of view. You're on your own to troubleshoot these problems.

We've found that one useful technique can be to use the old command-line Telnet client. Keep in mind that WS-MAN is just HTTP, and HTTP – like many Internet protocols – is just sending text back and forth, more or less exactly like Telnet. HTTP has specific text it sends and looks for, but the actual transmission is old-school Telnet. So we'll run something like `telnet dc01 5985` just to see if we can connect. A blank screen is normal: Hit Ctrl+C to break out, and you'll see an HTTP "Bad Request" error. That's fine – it means you got through. That confirms the computer name, the IP address, the port, and everything else "low-level."

Connection Problem: No Permissions

This can be a bit of a tricky problem, because you need to be an Administrator to enable a diagnostics trace. On the other hand, WinRM is usually quite clear when you can't connect because your account doesn't have permission to the endpoint: "Access Denied" is the error message, and that's pretty straightforward.

But you can also log on as an Administrator (or open a shell under Administrator credentials), enable a trace, and then have the other user (or your other user account) try whatever it is they're trying. Go back in as Administrator and disable the trace, then examine the log. Figure 4.14 shows what you're looking for.

```
 1840 4/14/2012 4:18:53 PM An error was encountered while processing an
                           operation.
                           Error Code: 5
                           Error String:<f:WSManFault xmlns:f="http://schemas.mi
                           crosoft.com/wbem/wsman/1/wsmanfault" Code="5"
                           Machine="dc01"><f:Message>Access is denied.
                           </f:Message></f:WSManFault>
  254 4/14/2012 4:18:53 PM Activity Transfer
  142 4/14/2012 4:18:53 PM WSMan operation CreateShell failed, error code 5
32786 4/14/2012 4:18:53 PM Runspace Id 0d91c610-3c82-4b15-8858-76d833a013a3.
                           Callback received for WSMan Create Shell
 1840 4/14/2012 4:18:53 PM An error was encountered while processing an
                           operation.
                           Error Code: 122
                           Error String:<f:WSManFault xmlns:f="http://schemas.mi
                           crosoft.com/wbem/wsman/1/wsmanfault" Code="122"
                           Machine="C3096161287.AD2008R2.loc"><f:Message>The
                           data area passed to a system call is too small.
                           </f:Message></f:WSManFault>
  319 4/14/2012 4:18:53 PM Getting message for error code 5 completed
                           successfully. The languageCode parameter was: en-US
 8196 4/14/2012 4:18:53 PM Modifying activity Id and correlating
12039 4/14/2012 4:18:53 PM Modifying activity Id and correlating
32784 4/14/2012 4:18:53 PM Runspace Id: 0d91c610-3c82-4b15-8858-76d833a013a3
                           Pipeline Id: 00000000-0000-0000-0000-000000000000.
                           WSMan reported an error with error code: 5.
                            Error message: Connecting to remote server dc01
                           failed with the following error message : Access is
                           denied. For more information, see the
                           about_Remote_Troubleshooting Help topic.
                            StackTrace:
32776 4/14/2012 4:18:53 PM Runspace Id: 0d91c610-3c82-4b15-8858-76d833a013a3
                           Pipeline Id: 00000000-0000-0000-0000-000000000000.
                           WSMan reported an error with error code: 5.
                            Error message: Connecting to remote server dc01
                           failed with the following error message : Access is
                           denied. For more information, see the
                           about_Remote_Troubleshooting Help topic.
```

Figure 4.14 "Access Denied" in the diagnostics log

The log data just after that will show you the user account that was used to try and create the connection (AD2008R2\SallyS, in our example, which is why the command failed – she's not an Administrator). A quick check with Get-PSSessionConfiguration on the remote machine will confirm the permissions on whatever Remoting endpoint you're attempting to connect to. Also, as shown in figure 4.15, we've found that running Set-PSSessionConfiguration can be useful. Provide the –Name of the endpoint you're checking, and add –ShowSecurityDescriptorUI. That will let you confirm the endpoint's permissions in a friendlier GUI form – and you can modify it right there if need be.

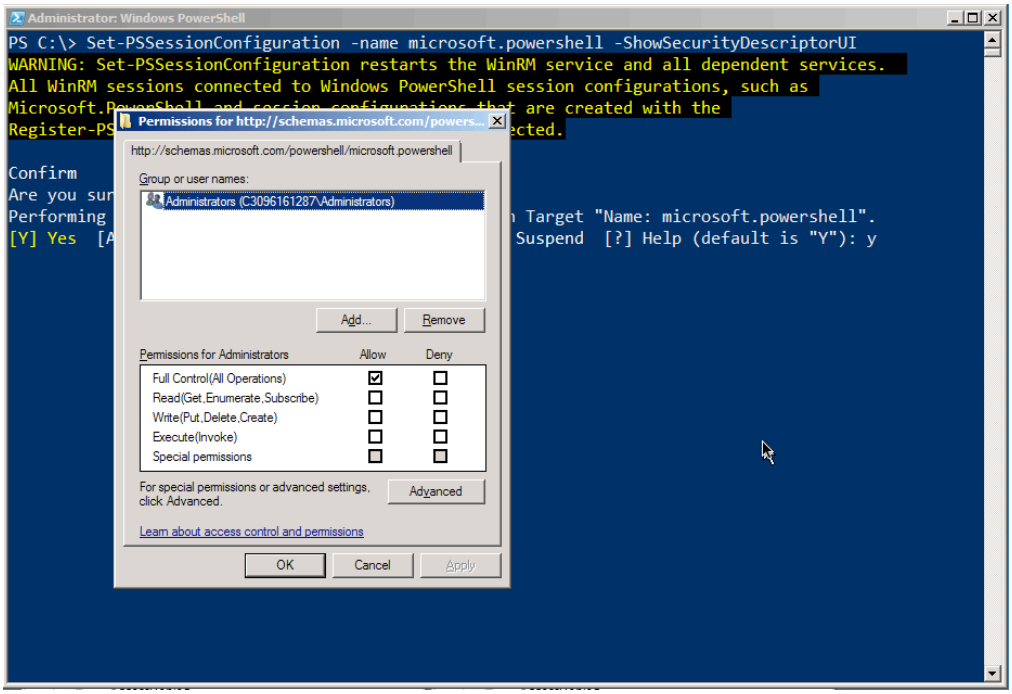


Figure 4.15 Checking an endpoint's permissions using Set-PSSessionConfiguration

Connection Problem: Untrusted Host

Figure 4-16 shows the connection we're trying to make: From the client in the AD2008R2 domain to a standalone computer that isn't part of a domain.

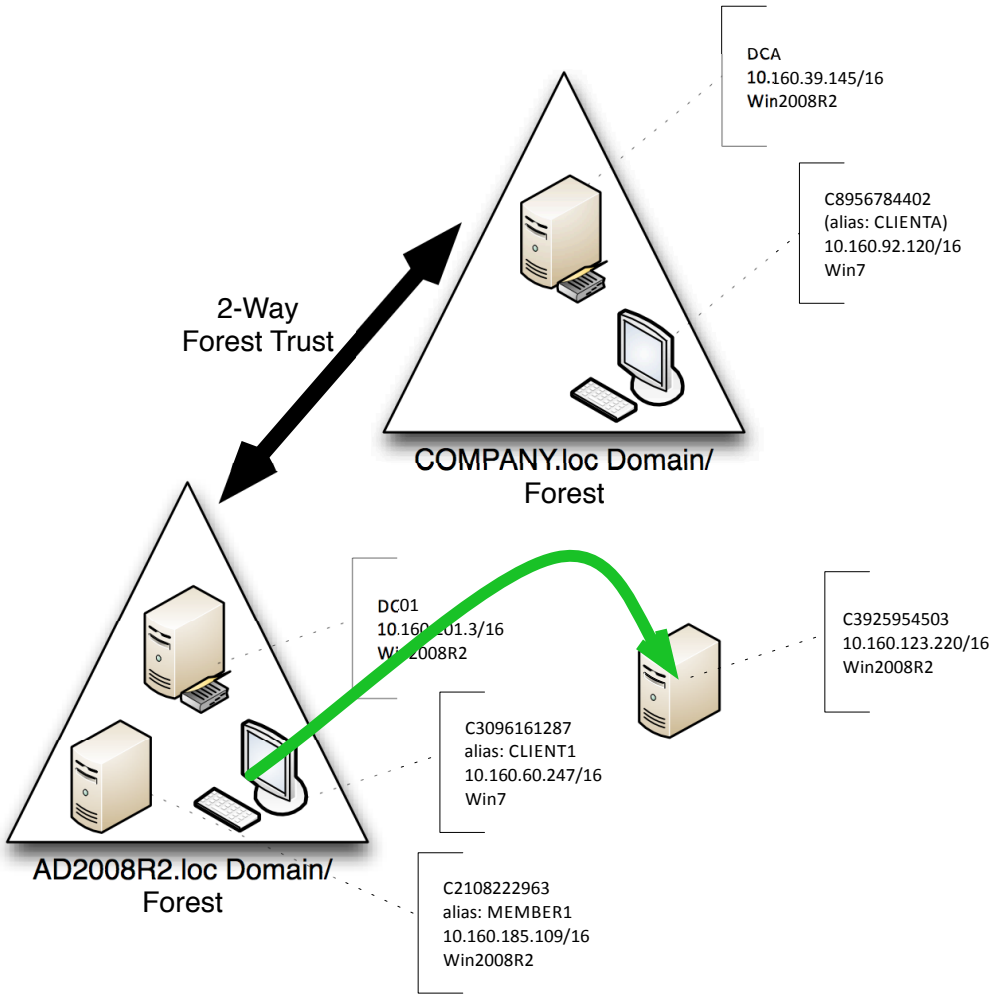


Figure 4.16 Attempted connection for this scenario

As shown in figure 4.17, the error comes quickly, even though we've provided a valid credential. The problem is that we're in a situation where WinRM can't get the mutual authentication it wants; part 2 of this guide covers solutions for fixing the problem. But what does the problem look like in the diagnostics log?

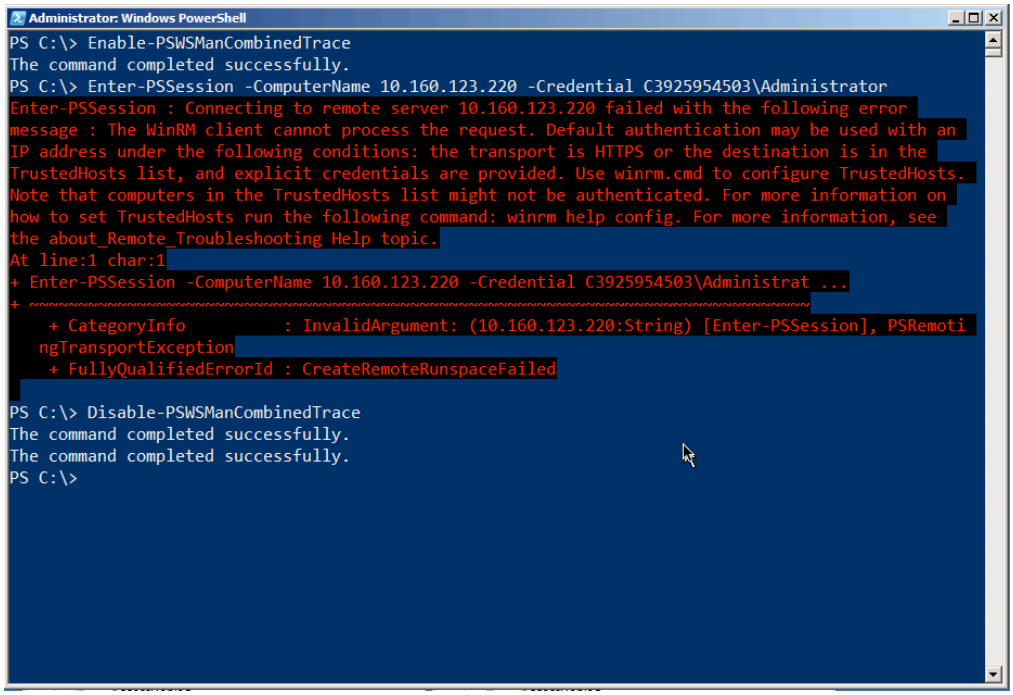


Figure 4.18 shows that WinRM still sends its initial salvo of traffic to the server. It's when the reply comes back that the client realizes it can't authenticate this server, and the error is generated. What you see in the log is pretty much what shows up in the shell, verbatim.

```
                         yWIWY3TMyPjwvTVM+PC9PYmo+PC9Fbj48RW4+PEkzMiBOPSJLZXki
                         PjM8L0kzMj48T2JqIE49IlZhbHVlIiBSZWZJZD0iMTkiPjxNUz48U
                         yBOPSJUIj5TeXN0ZW0uTWFuYWdlbWVudC5BdXRvbWF0aW9uLkhvc3
                         QuQ29vcmRpbmF0ZXM8L1M+PE9iaiBOPSJWIiBSZWZJZD0iMjAiPjx
                         NUz48STMyIE49IngiPjA8L0kzMj48STMyIE49InkiPjA8L0kzMj48
                         L01TPjwvT2JqPjwvTVM+PC9PYmo+PC9Fbj48RW4+PEkzMiBOPSJLZ
                         XkiPjI8L0kzMj48T2JqIE49IlZhbHVlIiBSZWZJZD0iMjEiPjxNUz
                         48UyBOPSJUIj5TeXN0ZW0uTWFuYWdlbWVudC5BdXRvbWF0aW9uLkh
                         vc3QuQ29vcmRpbmF0ZXM8L1M+PE9iaiBOPSJWIiBSZWZJZD0iMjIi
                         PjxNUz48STMyIE49IngiPjA8L0kzMj48STMyIE49InkiPjM8L0kzM
                         j48L01TPjwvT2JqPjwvTVM+PC9PYmo+PC9Fbj48RW4+PEkzMiBOPS
                         JLZXkiPjE8L0kzMj48T2JqIE49IlZhbHVlIiBSZWZJZD0iMjMiPjx
                         NUz48UyBOPSJUIj5TeXN0ZW0uQ29uc29sZUNvbG9yPC9TPjxJMzIg
                         Tj0iViI+NTwvSTMyPjwvTVM+PC9PYmo+PC9Fbj48RW4+PEkzMiBOP
                         SJLZXkiPjA8L0kzMj48T2JqIE49IlZhbHVlIiBSZWZJZD0iMjQiPj
                         xNUz48UyBOPSJUIj5TeXN0ZW0uQ29uc29sZUNvbG9yPC9TPjxJMzI
                         gTj0iViI+NjwvSTMyPjwvTVM
  779 4/14/2012 4:33:38 PM SOAP [client sending index 6 of 6 total chunks (289
                         bytes)] +PC9PYmo+PC9Fbj48L0RDVD48L09iaj48L01TPjwvT2Jq
                         PjxCIE49Il9pc0hvc3ROdWxsIj5mYWxzZTwvQj48QiBOPSJfaXNIb
                         3N0VUlOdWxsIj5mYWxzZTwvQj48QiBOPSJfaXNIb3N0UmF3VUlOdW
                         xsIj5mYWxzZTwvQj48QiBOPSJfdXNlUnVuc3BhY2VIb3N0Ij5mYWx
                         zZTwvQj48L01TPjwvT2JqPjwvTVM+PC9PYmo+</creationXml></
                         rsp:Shell></s:Body></s:Envelope>
 1840 4/14/2012 4:33:38 PM An error was encountered while processing an
                         operation.
                         Error Code: 2150859195
                         Error String:<f:WSManFault xmlns:f="http://schemas.mi
                         crosoft.com/wbem/wsman/1/wsmanfault"
                         Code="2150859195"
                         Machine="C3096161287.AD2008R2.loc"><f:Message>The
                         WinRM client cannot process the request. Default
                         authentication may be used with an IP address under
                         the following conditions: the transport is HTTPS or
                         the destination is in the TrustedHosts list, and
                         explicit credentials are provided. Use winrm.cmd to
                         configure TrustedHosts. Note that computers in the
                         TrustedHosts list might not be authenticated. For
                         more information on how to set TrustedHosts run the
                         following command: winrm help config.
                         </f:Message></f:WSManFault>
   12 4/14/2012 4:33:38 PM WSMan shell creation failed, error code 2150859195
32786 4/14/2012 4:33:38 PM Runspace Id 3a582e47-bc6a-4819-aae0-5a5bcc0b487b.
                         Callback received for WSMan Create Shell
 1840 4/14/2012 4:33:38 PM An error was encountered while processing an
                         operation
```

Figure 4-19 shows a good second step to take: Run Test-WSMan. Provide the same computer name or IP address, but leave off the –Credential parameter. The cmdlet can at least tell you that WS-MAN and WinRM are up and running on the remote computer, and what version they're running. That at least narrows the problem down to one of authentication: Either your

permissions (which would have resulted in an "Access Denied") or the mutual authentication component of Remoting.

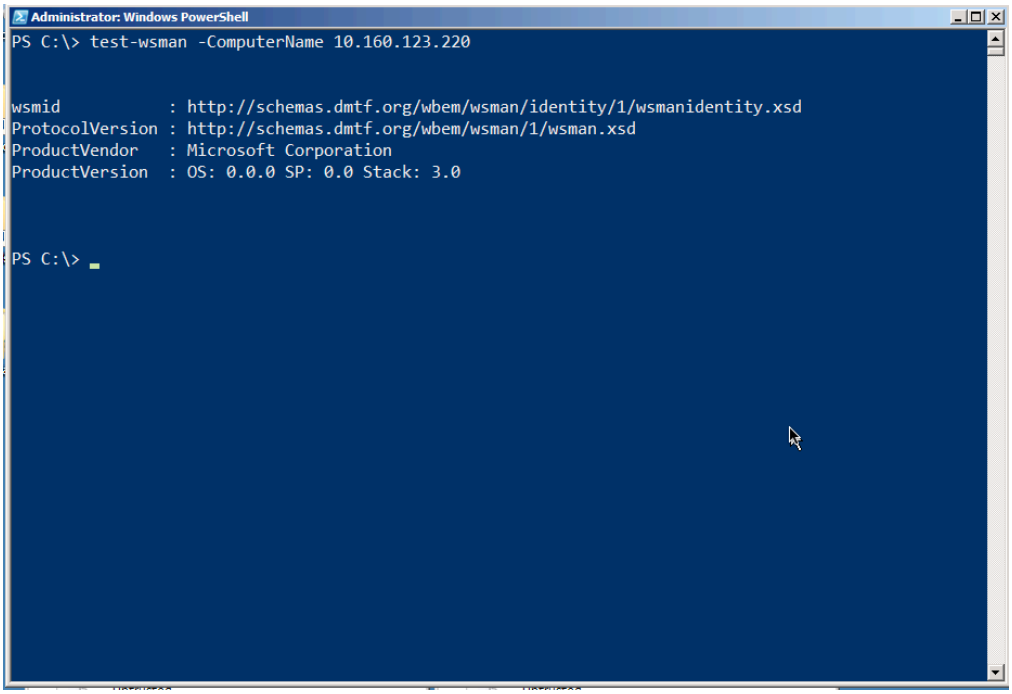


Figure 4.19 Test-WSMan is kind of like a "ping" for Remoting

> **Note** You'll see substantially the same behavior when you attempt to connect using HTTPS (the –UseSSL switch on the various Remoting commands), and the remote machine's SSL certificate name doesn't match the name you used in your command. The error message is unambiguous both on-screen and in the log, and we discuss solutions in part 2 of the guide.

Summary

So why did we bother going through the logs when, in most of our examples, the logs simply echoed what was on the screen? Simple: As PowerShell becomes embedded in more and more GUI applications, you might not always have a console, with its nice error messages, to rely upon. What you can do, however, is use the console to start a trace, run whatever GUI app is failing, and then dig into the log to see if you find some of the signs we've shown you here.

Session Management

When you create a Remoting connection between two machines, you're creating – in PowerShell terminology – a *session*. There are an incredible number of options that can be applied to these sessions, and in this portion of the guide we'll walk you through them.

Ad-Hoc vs. Persistent Sessions

When you use a Remoting command – primarily Invoke-Command or Enter-PSSession – and specify a computer name by using their –ComputerName parameter, you're creating an *ad-hoc session*. Basically, PowerShell just brings up a session, utilizes it, and then tears it down, all automatically.

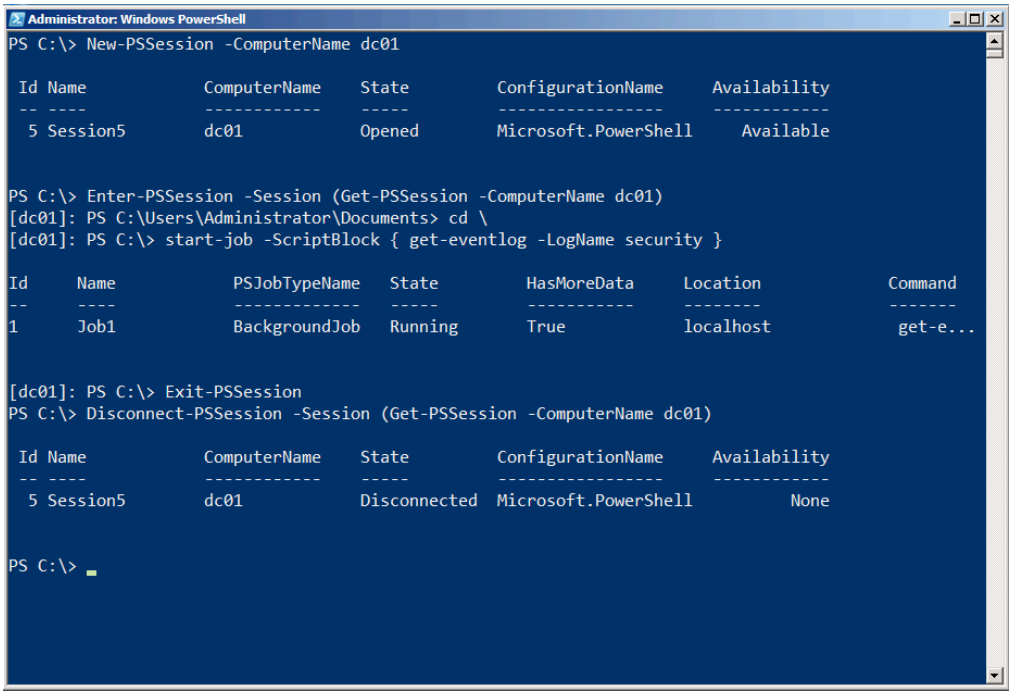
Alternately, you can use New-PSSession to explicitly create a new session, which can then be utilized by passing the session to the –Session parameter of Invoke-Command, Enter-PSSession, and numerous other Remoting-aware commands. When you manually create a session, it's up to you to get rid of it when you're done with it. However, if you have a session open and close your copy of PowerShell, that session is automatically removed for you – so you're not leaving anything hanging around that needs to be cleaned up.

Disconnecting and Reconnecting Sessions

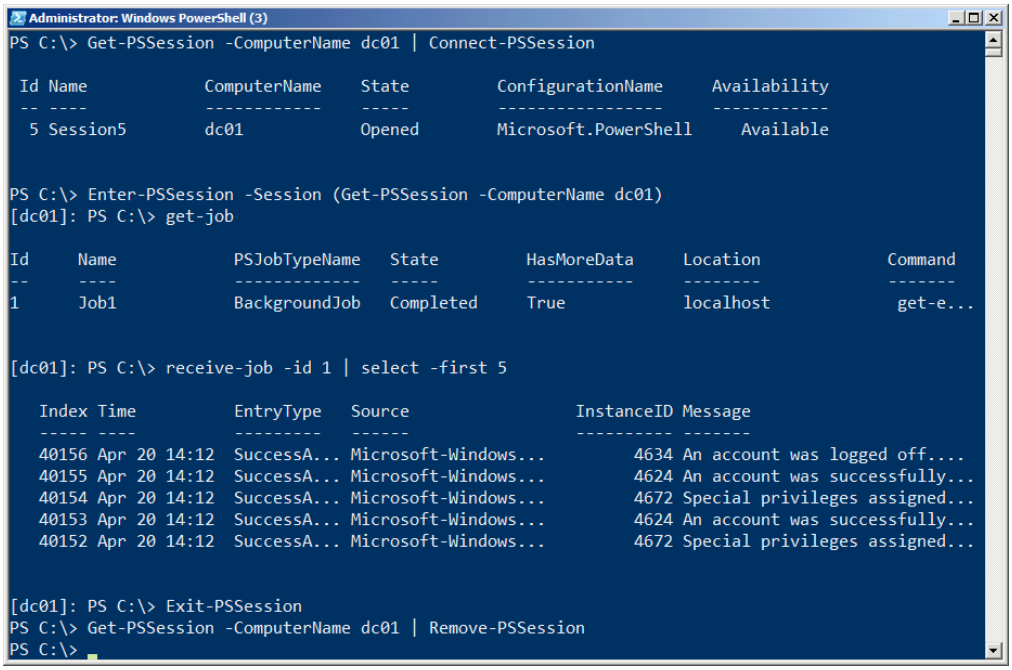
In PowerShell v3, you can disconnect and reconnect sessions by using Disconnect-PSSession and Connect-PSSession. These commands each accept a session object, which you'd usually create with New-PSSession.

A disconnected session leaves a copy of PowerShell up and running on the remote computer. This is a good way to get it to run some long-running task, disconnect, and then reconnect later to check up on it. You can even disconnect a session on one computer, move to another computer, and reconnect to that session (although you can't connect to someone else's disconnect session; you're limited to reconnecting to your own).

For example, figure 5.1 shows a session being created from a client to a server. The session is then given a task to perform as a background job, and then the session is disconnected. It's important to note that the command, and the background job, are on the server (DC01), not the client.



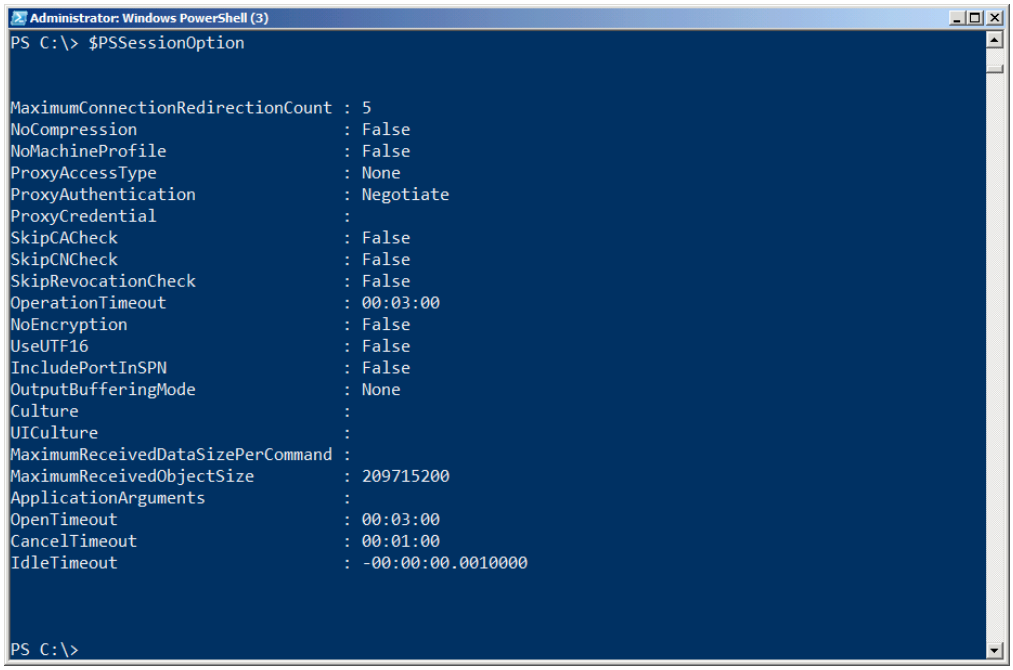
In figure 5.2, we've moved to a different machine. We're logged on, and running PowerShell, as the same user that we were on the previous client computer. We retrieve the session from the remote computer, and then reconnect it. We then enter the newly reconnected session, display that background job, and receive some results from it. Finally, we exit the remote session and shut it down via Remove-PSSession.



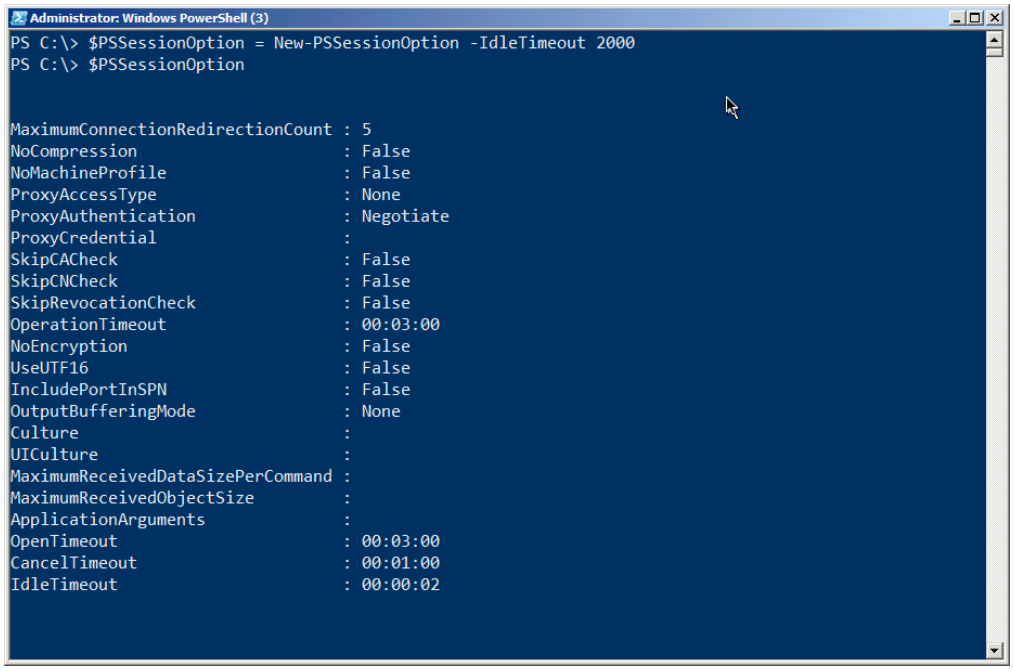
Obviously, disconnected sessions can present something of a management concern, because you're leaving a copy of PowerShell up and running on a remote machine – and you're doing so in a way that makes it difficult for someone else to even see you've done it! That's where session options come into play.

Session Options

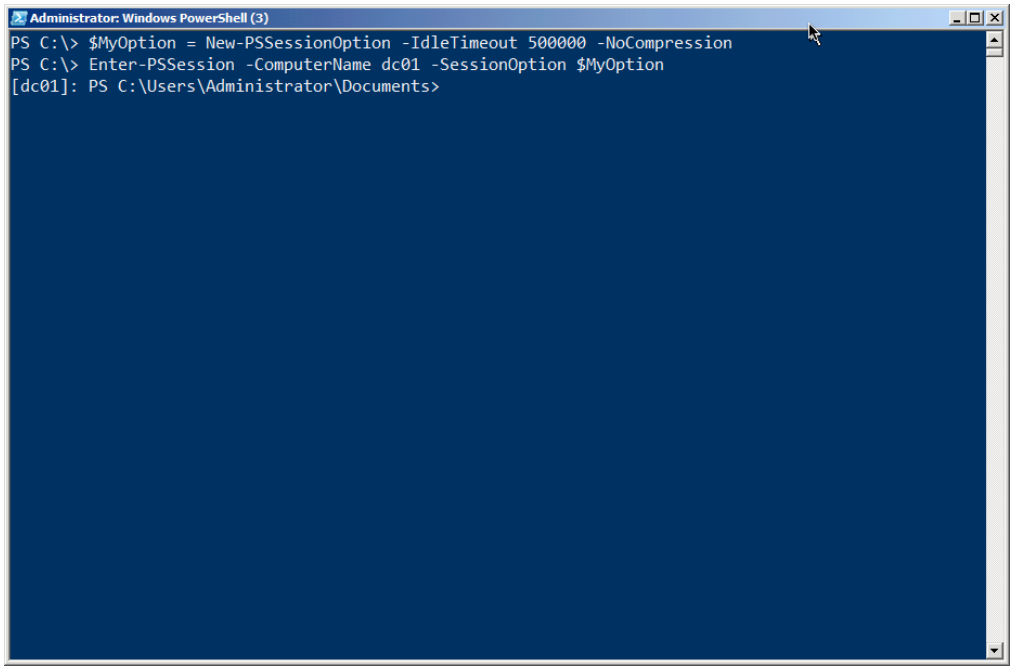
Whenever you run a Remoting command that creates a session – whether persistent or ad-hoc – you have the option of specifying a –SessionOption parameter, which accepts a PSSessionOption object. The default option object is used if you don't specify one, and that object can be found in the built-in $PSSessionOption variable. It's shown in figure 5.3.



As you can see, this specifies a number of defaults, including the operation timeout, idle timeout, and other options. You can change these by simply creating a new session option object and assigning it to $PSSessionOption; note that you need to do this in a profile script if you want your changes to become the new default every time you open a new copy of PowerShell. Figure 5.4 shows an example.



Of course, a 2-second idle timeout probably isn't very practical (and in fact won't work – you must specify at least a 60-second timeout in order to use the session object at all), but you'll note that you only need to specify the option parameters that you want to change – everything else will go to the built-in defaults. You can also specify a unique session option for any given session you create. Figure 5.5 shows one way to do so.



By specifying intelligent values for these various options, you can help ensure that disconnected sessions don't hang around and run forever and ever. A reasonable idle timeout, for example, ensures that the session will eventually close itself, even if an administrator disconnects from it and subsequently forgets about it. Note that, when a session closes itself, any data within that session – including background job results – will be lost. It's probably a good idea to get in the practice of having data saved into a file (by using Export-CliXML, for example), so that an idle session doesn't close itself and lose all of your work.

PowerShell, Remoting, and Security

Although PowerShell Remoting has been around since roughly 2010, many administrators and organizations are unable to take advantage of it, due in large part to outdated or uninformed security and risk avoidance policies. This chapter is designed to help address some of those by providing some honest technical detail about how these technologies work. In fact, they present significantly less risk than many of the management and communications protocols already in widespread use – those older protocols benefit primarily from being "grandfathered" into policies and never closely examined.

Neither PowerShell nor Remoting are a "Back Door" for Malware

This is a major misconception. Keep in mind that, by default, PowerShell does not execute scripts. When it does so, it can only execute commands that the executing user has permission to run – it does not execute anything under a super-privileged account, and it bypasses neither existing permissions nor security. In fact, because PowerShell is based upon .NET, it's unlikely any malware author would even bother to utilize PowerShell. Such an attacker could simply call on .NET Framework functionality directly, and much more easily.

By default, PowerShell Remoting enables only Administrators to even connect, and once connected they can only run commands they have permission to run – with no ability to bypass permissions or underlying security. Unlike past tools which ran under a highly-privileged account (such as LocalSystem), PowerShell Remoting executes commands by impersonating the user who submitted the commands.

Bottom line: Because of the way it works, PowerShell Remoting does not allow any user, authorized or not, to do anything that they could not do through a dozen other means – including logging onto the console. Whatever protections you have in place to prevent those kinds of attacks (such as appropriate authorization and authentication mechanisms) will also protect PowerShell and Remoting. If you allow Administrators to log on to server consoles – either physically or via Remote Desktop – you have far greater security exposure than you do through PowerShell Remoting.

Further, PowerShell offers a better opportunity to restrict even Administrators. A Remoting *endpoint* (or *session configuration*) can be modified to allow only specified users to connect to it. Once connected, the endpoint can further restrict the commands that those users can execute. This provides a much better opportunity for delegated administration. Rather than

having Administrators log onto consoles and do whatever they please, you can have them connect to restricted, secured endpoints and only complete those specific tasks that the endpoint permits.

PowerShell Remoting is Not Optional

As of Windows Server 2012, PowerShell Remoting is enabled by default and is *mandatory* for server management. Even when running a graphical management console locally on a server, the console still "goes out" and "back in" via Remoting to accomplish its tasks. Without Remoting, server administration is impossible. Organizations are therefore well-advised to start *immediately* finding a way to include Remoting in their permitted protocols. Otherwise, critical services *will not be able to be managed,* even through Remote Desktop or directly on the server console.

This approach actually helps better secure the data center. Because local administration is *exactly* the same as remote administration (via Remoting), there's no longer any reason to physically or remotely access server consoles. The consoles can thus remain more locked down and secured, and Administrators can stay out of the data center entirely.

Remoting Does Not Transmit or Store Credentials

By default, Remoting uses Kerberos, an authentication protocol that does not transmit passwords across the network. Instead, Kerberos relies on passwords as an encryption key, ensuring that passwords remain safe. Remoting can be configured to use less-secure authentication protocols (such as Basic), but can also be configured to require certificate-based encryption for the connection.

Further, Remoting never stores credentials in any persistent storage by default. A Remote machine never has access to a user's credentials; it has access only to a delegated security token (a Kerberos "ticket"). That is stored in volatile memory which cannot, by OS design, be written to disk – even to the OS page file. The server presents that token to the OS when executing commands, causing the command to be executed with the original invoking user's authority – and nothing more.

Remoting Uses Encryption

Most Remoting-enabled applications apply their own encryption to their application-level traffic sent over Remoting. However, Remoting can also be configured to use HTTPS (certificate-encrypted connections), and can be configured to make HTTPS mandatory. This encrypts the entire channel using high-level encryption, while also ensuring mutual authentication of both client and server.

Remoting is Security-Transparent

As stated, Remoting neither adds anything to, nor takes anything away from, your existing security configuration. Remote commands are executed using the delegated credentials of whatever user invoked the commands, meaning they can only do what they have permission to do – and what they could presumably do through a half-dozen other tools anyway. Whatever auditing you have in place in your environment cannot be bypassed by Remoting. Unlike many past "remote execution" solutions, Remoting does not operate under a single "super-privileged" account unless you expressly configure it that way (which requires several steps and cannot possibly by accomplished accidentally, as it requires the creation of custom endpoints).

Remember: Anything someone can do via Remoting, they can already do in a half-dozen other ways. Remoting simply provides a more consistent, controllable, and scalable means of doing so.

Remoting is Lower Overhead

Unlike Remote Desktop Connection (RDC, which many Administrators currently use to manage remote servers), Remoting is very low-overhead. It does not require the server to spin up an entire graphical operating environment, impacting server performance and memory management. Remoting is also more scalable, enabling authorized users (mainly Administrators in most cases) to execute commands against multiple servers at once – which improves consistency and reduces error, while also speeding up response times and lowering administrative overhead.

Remoting is Microsoft's way forward. To not use Remoting is to deliberately attempt to use Windows in a way that it was explicitly designed not to do. You will reduce, not improve your security, while also increasing operational overhead, enabling greater instance of human error, and reducing server performance. Microsoft Administrators have for decades been toiling under an operational paradigm that was wrong-headed and short-sighted; Remoting is finally delivering to Windows the administrative model that every other network operating system has used for years, if not decades.

Remoting Uses Mutual Authentication

Unlike nearly every other remote management technique out there – including tools like PSExec and even, under some circumstances, Remote Desktop, PowerShell Remoting by default requires mutual authentication. The user attempting to connect to a server is authenticated and known; the system also ensures that the server connected to is the intended server and not an imposter. This provides far better security than past techniques, while also helping to reduce error – you can't "accidentally log on to the wrong console" as you could if you just walked into the data center.

Summary

At this point, denying PowerShell Remoting is like denying Ethernet: It's ridiculous to think you'll successfully operate your environment without it. For the first time, Microsoft has provided a supported, official, baked-in technology for remote server administration that does not use elevated credentials, does not store credentials in any way, that supports mutual authentication, and that is complete security-transparent. This is the administration technology we should have had all along; moving to it will only make your environment more manageable and more secure, not less.

Configuring Remoting via GPO

PowerShell's about_remote_troubleshooting provides a good set of steps for configuring basic Remoting functionality via Group Policy objects (GPOs). Running Enable-PSRemoting also reveals some useful details, such as the four main configuration. In this section, we'll cover these main configuration steps.

> **Note** None of this is necessary on Windows Server 2012 and later versions of the server OS. Remoting is enabled by default on those, and shouldn't be turned off, as many of the native management tools (including GUI consoles like Server Manager) depend upon Remoting.

GPO Caveats

One thing to keep in mind is that GPOs can only create configuration changes; they can't necessarily change the active state of the computer. In other words, while a GPO can configure a service's start mode to "Automatic," it can't *start* the service. That'll happen automatically when the computer is restarted. It isn't so much that a restart is needed, just that the computer only *starts* services after booting. So in many cases, the changes you make with a GPO (with regard to Remoting) won't actually take affect until the next time the affected computers are restarted, because in most cases the computer only *looks at the configuration* at boot time. Just be aware of that.

Also, everything in this section assumes that PowerShell is already installed on the target computers – something that can also be accomplished with a GPO or other software deployment mechanism, but not something we're going to cover here. Note that most of this section should apply to either PowerShell v2 or v3; we're going to run through the examples using v2 on a Windows 7 client computer belonging to a Windows Server 2008 R2 domain.

> **Note** Some of the GPO settings we'll be reviewing became available in Windows 2008 and Windows 2008 R2, but you should be able to install the necessary administrative templates into any domain controller. The Windows 7 (and later versions) Remote Server Administration Toolkit (RSAT) contains the necessary templates.

We don't know for sure that the GPO configuration steps need to be accomplished in the order we present them; in most cases, we expect you'll do them all at once in a single GPO, so it won't matter. We're taking them step-by-step in this order so that we can check the individual results along the way.

Allowing Automatic Configuration of WinRM Listeners

As explained earlier in this guide, the WinRM service sets up one or more *listeners* to accept incoming traffic. Running Enable-PSRemoting, for example, sets up an HTTP listener, and we've covered how to set up an HTTPS listener in addition to, or instead of, that default one.

You'll find this setting under: Computer Configuration\Administrative Templates\Windows Components\Windows Remote Management (WinRM)\WinRM Service. Enable the policy, and specify the IPv4 and IPv6 filters, which determine which IP addresses listeners will be configured on. You can use the * wildcard to designate all IP addresses, which is what we've done in Figure 7.1.

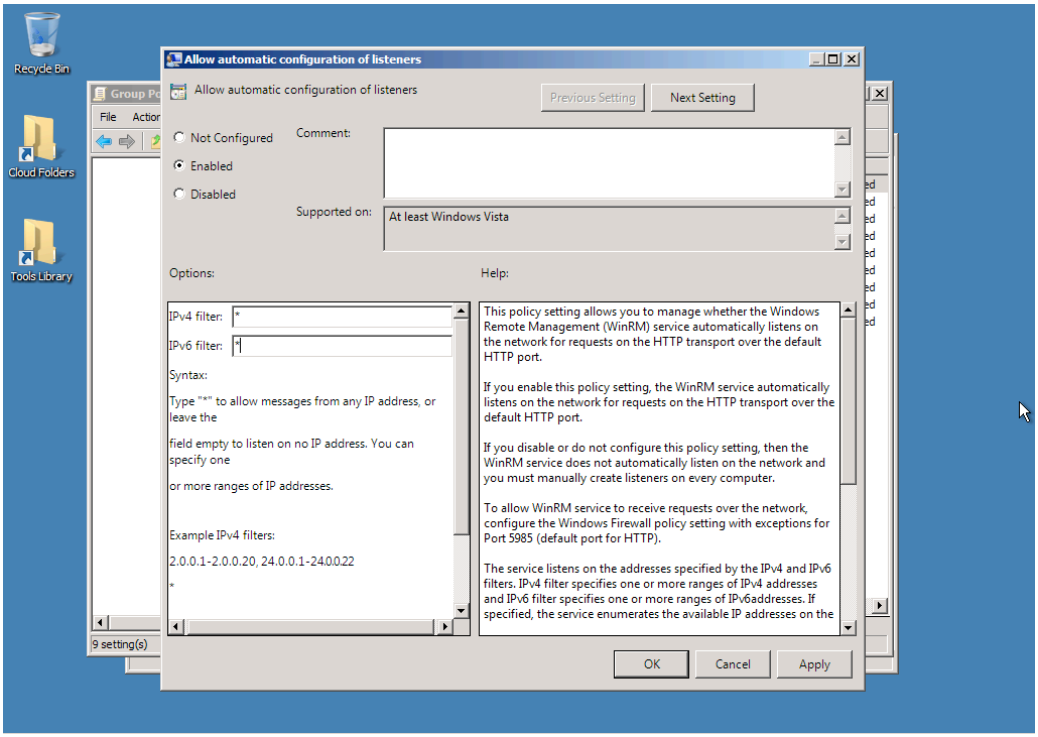


Figure 7.1 Enabling automatic configuration of WinRM listeners

Setting the WinRM Service to Start Automatically

This service is set to start automatically on newer server operating systems (Windows Server 2003 and later), but not on clients. So this step will only be required for client computers. Again, this won't *start* the service, but the next time the computer restarts, the service will start automatically.

Microsoft suggests accomplishing this task by running a PowerShell command – which does *not* require that Remoting be enabled in order to work:

```
Set-Service WinRM –computername $servers –startup Automatic
```

You can populate $servers any way you like, so long as it contains strings that are computer names, and so long as you have Administrator credentials on those computers. For example, to grab every computer in your domain, you'd run the following (this assumes PowerShell v2 or v3, on a Windows 7 computer with the RSAT installed):

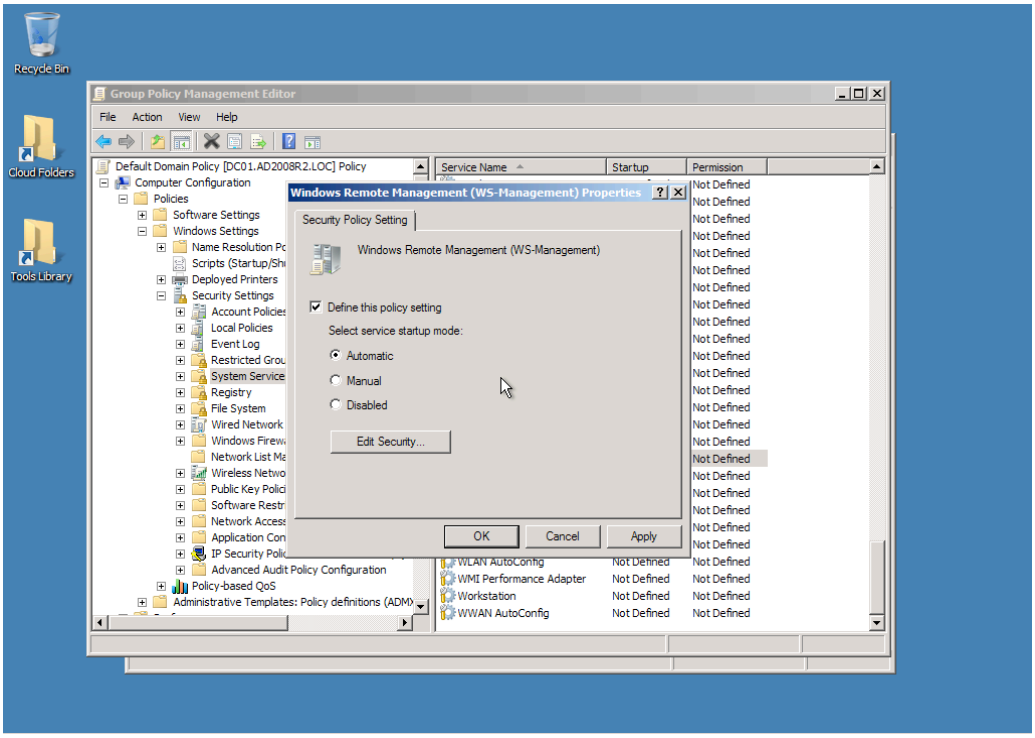
```
Import-Module ActiveDirectory

$servers = Get-ADComputer –filter * | Select –expand name
```

Practically speaking, you'll probably want to limit the number of computers you do at once by either specifying a –filter other than "*" or by specifying –searchBase and limiting the search to a specific OU. Read the help for Get-ADComputer to learn more about those parameters.

Note that Set-Service will return an error for any computers it couldn't contact, or for which the change didn't work, and then continue on with the next computer.

Alternately, you could configure this with a GPO. Under Computer Configuration\Windows Settings\Security Settings\System Services, look for "Windows Remote Management." Right-click it and set a startup mode of Automatic. That's what we did in figure 7.2.

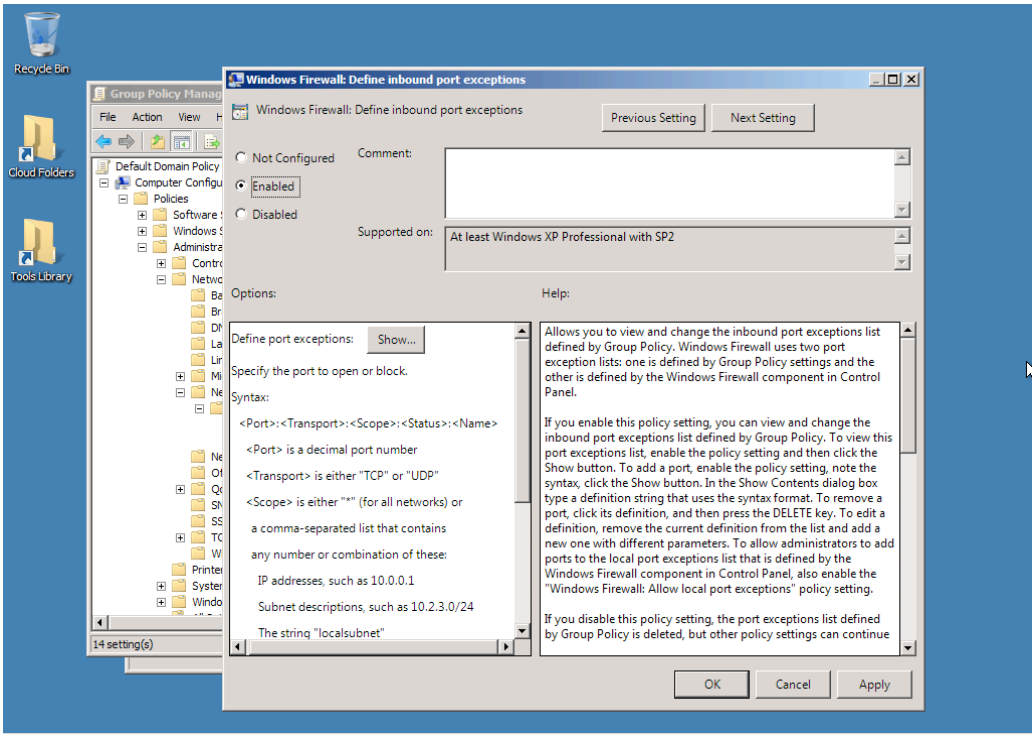


Creating a Windows Firewall Exception

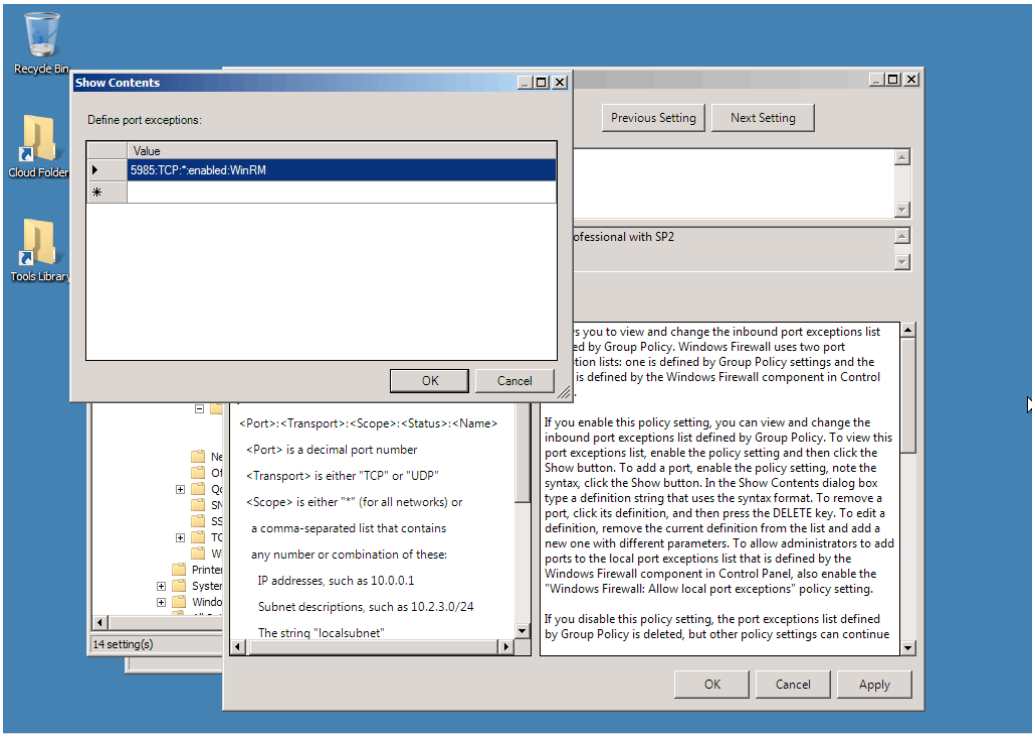
This step will be necessary on all computers where the Windows Firewall is enabled. We're assuming that you only want Remoting enabled in your Domain firewall profile, so that's all we're doing in our example. Obviously, you can manage whatever other exceptions you want in whatever profiles are appropriate for your environment.

You'll find one setting under Computer Configuration\Administrative Templates\Network\Network Connections\Windows Firewall\Domain Profile. Note that the "Windows Firewall: Allow Local Port Exceptions" policy simply allows local Administrators to configure Firewall exceptions using the Control Panel; it doesn't actually create any exceptions. That may be exactly what you want in some cases.

Instead, we went to the "Define inbound port exceptions" policy, and Enabled it, as shown in figure 7.3.



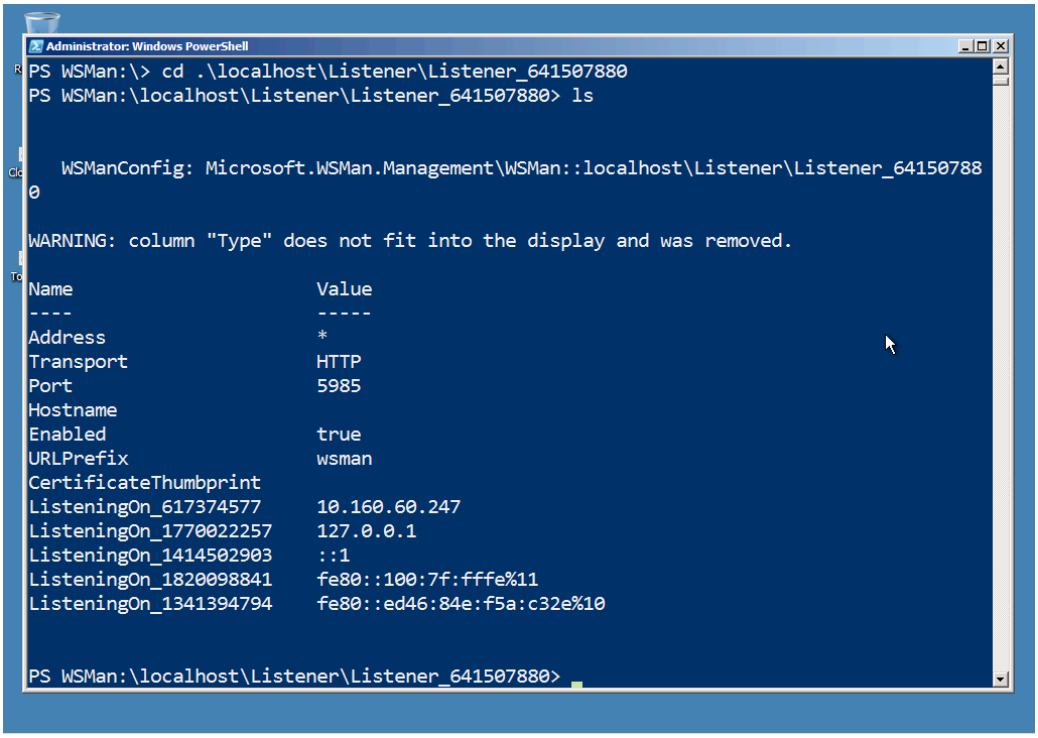
We then clicked “Show,” and added “5985:TCP:*:enabled:WinRM” as a new exception, as shown in figure 7.4.



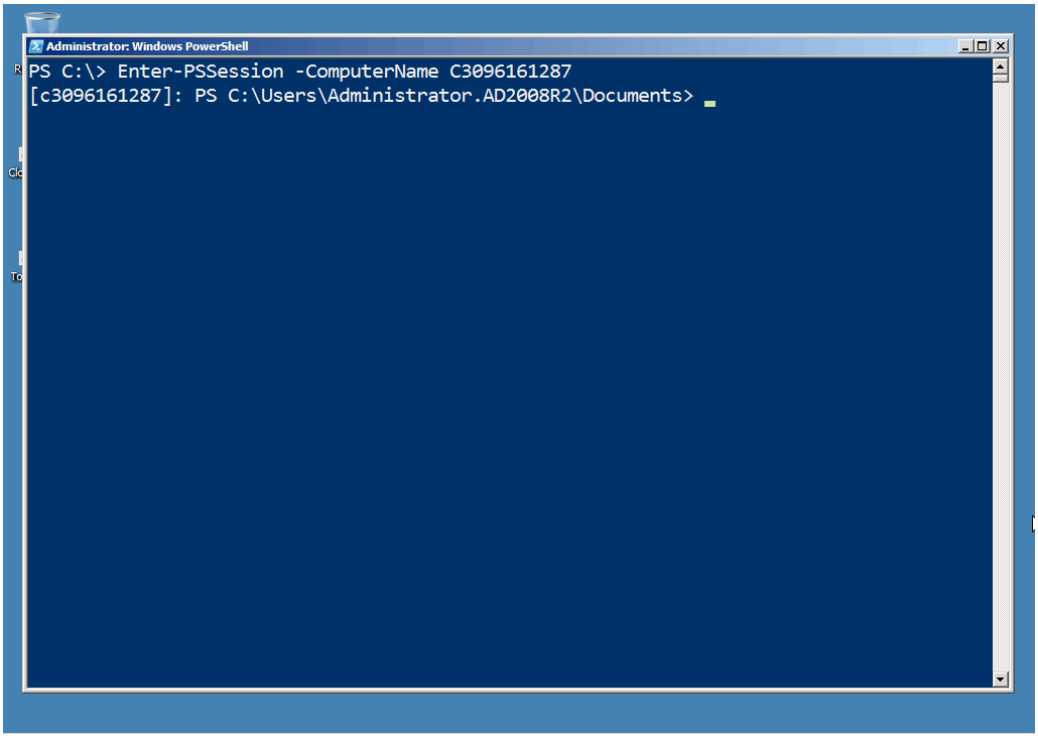
Give it a Try!

After applying the above GPO changes, we restarted our client computer. When the WinRM service starts, it checks to see if it has any configured listeners. When it finds that it doesn't, it should try and automatically configure one – which we've now allowed it to do via GPO. The Firewall exception should allow the incoming traffic to reach the listener.

As shown in figure 7.5, it seems to work. We've found the newly created listener!



Of course, the proof – as they say – is in the pudding. So we ran to another computer and, as shown in figure 7.6, were able to initiate an interactive Remoting session to our original client computer. We didn't configure anything except via GPO, and it's all working.



What You Can't Do with a GPO

You can't use a GPO to start the WinRM service, as we've already stated. You also can't create custom listeners via GPO, nor can you create custom PowerShell endpoints (session configurations). However, once basic Remoting is enabled via GPO, you can use PowerShell's Invoke-Command cmdlet to remotely perform those other tasks. You could even use Invoke-Command to remotely disable the default HTTP listener, if that's what you wanted.

Also keep in mind that PowerShell's WSMAN PSProvider can map remote computers' WinRM configuration into your local WSMAN: drive. That's why, by default, the top-level "folder" in that drive is "localhost;" so that there's a spot to add other computers, if desired. That offers another way to configure listeners and other Remoting-related settings.

The real key is to use GPO to get Remoting up and running in this basic form, which is what we've shown you how to do. From there, you can use Remoting itself to tweak, reconfigure, and modify the configuration.

Afterword

We hope you've found this guide to be useful! We want to re-emphasize that this is always going to be a work in progress; as people bring us material and suggestions, we'll incorporate that as best we can and publish a new edition, hosting the master copy at http://PowerShellBooks.com. We consider the PDF version to be the most authoritative, because it preserves the maximum amount of our formatting. Any other editions – EPUB, MOBI, and so forth – are created primarily for the convenience of you, our reader; we don't have the resources (unfortunately) to spend a lot of time tweaking those formats, so sometimes their presentation of this material may be sub-optimal.

We welcome your input! Both principle authors, Don Jones and Tobias Weltner, offer on-site PowerShell training classes. Don primarily covers the US, while Tobias works throughout Europe. Both can be reached in the "Ask the Experts" forums at PowerShell.com: http://powershell.com/cs/forums/default.aspx?GroupID=24. You can also reach Don (who coordinates updates to this guide) at http://ConcentratedTech.com/contact. Please submit technical questions to the appropriate "Ask the Experts" forum, not via that Contact page.