

Windows PowerShell[™] Scripting Guide



Ed Wilson

PUBLISHED BY

Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2008 by Ed Wilson

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2007941089

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, Microsoft Press, Active Directory, ActiveX, Excel, Internet Explorer, MSDN, MSN, Outlook, SQL Server, Visual Basic, Windows, Windows NT, Windows PowerShell, Windows Server, Windows Vista, and Zune are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Martin DelRe

Developmental Editor: Karen Szall

Project Editor: Denise Bankaitis and Michelle Goodman

Editorial Production: Custom Editorial Productions, Inc.

Technical Reviewer: Bob Hogan; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Cover: Tom Draper Design

Body Part No. X14-14922

Table of Contents

<i>Acknowledgments</i>	xvii
<i>Introduction</i>	xix
<i>Is This Book for Me?</i>	xix
<i>About the Companion CD</i>	xx
<i>System Requirements</i>	xxi
<i>Technical Support</i>	xxi
1 The Shell in Windows PowerShell	1
Installing Windows PowerShell	1
Verifying Installation with VBScript	1
Deploying Windows PowerShell	2
Interacting with the Shell	3
Introducing Cmdlets	5
Configuring Windows PowerShell	6
Creating a Windows PowerShell Profile	6
Configuring Windows PowerShell Startup Options	6
Security Issues with Windows PowerShell	7
Controlling the Execution of Cmdlets	7
Confirming Commands	9
Suspending Confirmation of Cmdlets	10
Supplying Options for Cmdlets	11
Working with Get-Help	12
Working with Aliases to Assign Shortcut Names to Cmdlets	15
Additional Uses of Cmdlets	16
Using the Get-ChildItem Cmdlet	17
Formatting Output	17
Using the Get-Command Cmdlet	24

 **What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

	Exploring with the Get-Member Cmdlet.	27
	Summary	31
2	Scripting Windows PowerShell.	33
	Why Use Scripting?	33
	Configuring the Scripting Policy	36
	Running Windows PowerShell Scripts.	39
	Use of Variables.	39
	Use of Constants.	40
	Using Flow Control Statements	41
	Adding Parameters to ForEach-Object	42
	Using the <i>Begin</i> Parameter	42
	Using the <i>Process</i> Parameter.	43
	Using the <i>End</i> Parameter.	43
	Using the <i>For</i> Statement	43
	Using Decision-Making Statements.	44
	Using <i>If ... Elseif ... Else</i>	45
	Using <i>Switch</i>	46
	Working with Data Types	49
	Unleashing the Power of Regular Expressions	53
	Using Command-Line Arguments	56
	Summary	58
3	Managing Logs	59
	Identifying the Event Logs	59
	Reading the Event Logs.	60
	Exporting to Text.	61
	Export to XML	62
	Perusing General Log Files	64
	Examining Multiple Logs	65
	Retrieving a Single Event Log Entry	66
	Searching the Event Log	68
	Filtering on Properties	69
	Selecting the Source.	69
	Selecting the Severity.	70
	Selecting the Message.	70

Managing the Event Log	71
Identifying the Sources.....	71
Modifying the Event Log Settings.....	71
Examining WMI Event Logs.....	75
Making Changes to the WMI Logging Level.....	76
Using the Windows Event Command-Line Utility	76
Writing to Event Logs.....	77
Creating a Source	77
Putting Cmdlet Output into the Log	78
Creating Your Own Event Logs	79
Summary.....	80
4 Managing Services	81
Documenting the Existing Services	81
Working with Running Services	82
Writing to a Text File.....	83
Writing to a Database.....	85
Setting the Service Configuration.....	94
Accepting Command-Line Arguments	97
Stopping Services	97
Performing a Graceful Stop	99
Starting Services	101
Performing a Graceful Start.....	102
Desired Configuration Maintenance	107
Verifying Desired Services Are Stopped.....	108
Reading a File to Check Service Status.....	109
Verifying Desired Services Are Running.....	110
Confirming the Configuration.....	110
Producing an Exception Report	111
Summary.....	113
5 Managing Shares.....	115
Documenting Shares.....	115
Documenting User Shares	122
Writing Shares to Text.....	125
Documenting Administrative Shares	126
Writing Share Information to a Microsoft Access Database.....	126

	Auditing Shares	130
	Modifying Shares	133
	Using Parameters with the Script	134
	Translating the Return Code	135
	Creating New Shares	137
	Creating Multiple Shares	141
	Deleting Shares	143
	Deleting Only Unauthorized Shares	145
	Summary	146
6	Managing Printing	147
	Inventorying Printers	147
	Querying Multiple Computers	148
	Logging to a File	150
	Writing to a Microsoft Access Database	152
	Reporting on Printer Ports	157
	Identifying Printer Drivers	163
	Installing Printer Drivers	165
	Installing Printer Drivers Found on Your Computer	165
	Installing Printer Drivers Not Found on Your Computer	167
	Summary	169
7	Desktop Maintenance	171
	Maintaining Desktop Health	171
	Inventorying Drives	171
	Writing Disk Drive Information to Microsoft Access	175
	Working with Partitions	179
	Matching Disks and Partitions	181
	Working with Logical Disks	184
	Monitoring Disk Space Utilization	188
	Logging Disk Space to a Database	192
	Monitoring File Longevity	196
	Monitoring Performance	199
	Using Performance Counter Classes	200
	Identifying Sources of Page Faults	204
	Summary	204

8	Networking	207
	Working with Network Settings	207
	Reporting Networking Settings	207
	Working with Adapter Configuration	212
	Filtering Only Properties that Have a Value	218
	Configuring Network Adapter Settings	223
	Detecting Multiple Network Adapters	223
	Writing Network Adapter Information to a Microsoft Excel Spreadsheet	224
	Identifying Connected Network Adapters	228
	Setting Static IP Address	230
	Enabling DHCP	235
	Configuring the Windows Firewall	239
	Reporting Firewall Settings	240
	Configuring Firewall Settings	241
	Summary	243
9	Configuring Desktop Settings	245
	Working with Desktop Configuration Issues	245
	Setting Screen Savers	245
	Auditing Screen Savers	246
	Listing Only Properties with Values	252
	Reporting Secure Screen Savers	256
	Managing Desktop Power Settings	263
	Changing the Power Scheme	269
	Summary	275
10	Managing Post-Deployment Issues	277
	Setting the Time	277
	Setting the Time Remotely	278
	Logging Results to the Event Log	283
	Configuring the Time Source	289
	Using the Net Time Command	290
	Querying the Registry for the Time Source	292
	Enabling User Accounts	297
	Creating a Local User Account	303
	Creating a Local User	303
	Creating a Local User Group	306

	Configuring the Screen Saver	309
	Renaming the Computer	316
	Shutting Down or Rebooting a Remote Computer	319
	Summary	323
11	Managing User Data	325
	Working with Backups	325
	Configuring Offline Files	328
	Enabling the Use of Offline Files	331
	Working with System Restore	340
	Retrieving System Restore Settings	340
	Listing Available System Restore Points	344
	Summary	347
12	Troubleshooting Windows	349
	Troubleshooting Startup Issues	349
	Examining the Boot Configuration	349
	Examining Startup Services	352
	Displaying Service Dependencies	355
	Examining Startup Device Drivers	360
	Investigating Startup Processes	365
	Investigating Hardware Issues	368
	Troubleshooting Network Issues	373
	Summary	377
13	Managing Domain Users	379
	Creating Organizational Units	379
	Creating Domain Users	382
	Modifying User Attributes	385
	Modifying General User Information	386
	Modifying the Address Tab	387
	Modifying the Profile Tab	388
	Modifying the Telephone Tab	389
	Modifying the Organization Tab	389
	Modifying a Single User Attribute	390
	Creating Users from a .csv File	393
	Setting the Password	394
	Enabling the User Account	394

Creating Domain Groups	395
Adding a User to a Domain Group	398
Adding Multiple Users with Multiple Attributes	400
Summary	404
14 Configuring the Cluster Service	405
Examining the Clustered Server	405
Reporting Cluster Configuration	411
Reporting Node Configuration	416
Querying Multiple Cluster Classes	420
Managing Nodes	431
Adding and Evicting Nodes	431
Removing the Cluster	437
Summary	442
15 Managing Internet Information Services	443
Enabling Internet Information Services Management	443
Reporting IIS Configuration	445
Reporting Site Information	445
Reporting on Application Pools	447
Reporting on Application Pool Default Values	451
Reporting Site Limits	454
Listing Virtual Directories	457
Creating a New Web Site	459
Creating a New Application Pool	464
Starting and Stopping Web Sites	467
Summary	471
16 Working with the Certificate Store	473
Locating Certificates in the Certificate Store	473
Listing Certificates	479
Locating Expired Certificates	483
Identifying Certificates about to Expire	488
Managing Certificates	492
Inspecting a Certificate	492
Importing a Certificate	497
Deleting a Certificate	501
Summary	507

17	Managing the Terminal Services Service	509
	Configuring the Terminal Service Installation	509
	Documenting Terminal Service Configuration	509
	Disabling Logons	513
	Modifying Client Properties	517
	Managing Users	521
	Enabling Users to Access the Server	524
	Configuring Client Settings	527
	Summary	539
18	Configuring Network Services	541
	Reporting DNS Settings	541
	Configuring DNS Logging Settings	548
	Reporting Root Hints	556
	Querying "A" Records	557
	Configuring DNS Server Settings	562
	Reporting DNS Zones	568
	Creating DNS Zones	571
	Managing WINS and DHCP	576
	Summary	581
19	Working with Windows Server 2008 Server Core	583
	Initial Configuration	583
	Joining the Domain	584
	Setting the IP Address	592
	Configuring the DNS Settings	597
	Renaming the Server	605
	Managing Windows Server 2008 Server Core	611
	Monitoring the Server	611
	Querying Event Logs	614
	Summary	617
A	Cmdlet Naming Conventions	619
B	ActiveX Data Object Provider Names	621
C	Frequently Asked Questions	623

D	Scripting Guidelines	631
	General Script Construction.....	631
	Include Functions in the Script that Calls the Function.....	631
	Use Full Cmdlet Names and Full Parameter Names.....	632
	Use Get-Item to Convert Path Strings to Rich Types	633
	General Script Readability	633
	Formatting Your Code	634
	Working with Functions	635
	Creating Template Files.....	637
	Writing Functions	637
	Creating and Naming Variables and Constants	638
E	General Troubleshooting Tips.....	639
	Index.....	643



What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Working with the Certificate Store

After completing this chapter, you will be able to:

- Locate specific certificates in the certificate store.
- List certificate stores.
- List certificates.
- Locate expired certificates.
- Import certificates.
- Delete certificates.



On the Companion Disc All the scripts used in this chapter are located on the CD that accompanies this book in the `\scripts\chapter16` folder.

Locating Certificates in the Certificate Store

A number of certificate stores reside on any Windows Vista or Windows Server 2008 computer. As certificates become more important, the ability to manage them becomes critical. One common problem with certificates is they aren't easily discovered. If you use the Certificate Manager Utility, as shown in Figure 16-1, you're confronted with a confusing array of folders with very little explanation and names that aren't intuitive.

On the other hand, if you use the certificate provider from within Windows PowerShell, the command is easy to use and does not cause the ubiquitous User Account Control dialog box to appear.

You can use the `Get-ChildItem` cmdlet to retrieve information about the certificate store locations:

```
Get-ChildItem cert:\
```

After using this command, you'll receive information about both the `CurrentUser` certificate store location, and the `LocalMachine` certificate store location. This information is displayed here:

```
Location      : CurrentUser
StoreNames    : {UserDS, AuthRoot, CA, Trust...}

Location      : LocalMachine
StoreNames    : {AuthRoot, CA, Trust, Disallowed...}
```

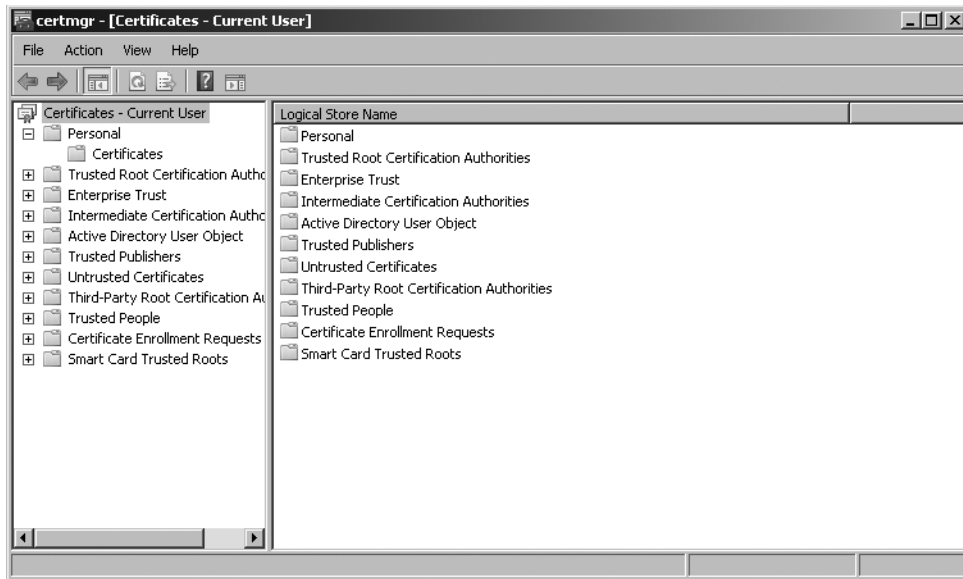


Figure 16-1 The Certificate Manager can be confusing because of the large number of folders.

While locating the CurrentUser certificate store location may be of interest, it becomes much more important to be able to work with the various certificate stores under either the current CurrentUser or the LocalMachine. To identify the various certificate stores for the CurrentUser, use the following command:

```
Get-ChildItem cert:\CurrentUser
```

After you receive a listing of the certificate stores under the CurrentUser, obtain a listing that is similar to the following. The actual certificate stores displayed will depend upon which applications are installed and which certificate stores have been configured:

```
Name : UserDS
Name : AuthRoot
Name : CA
Name : Trust
Name : Disallowed
Name : My
Name : Root
Name : TrustedPeople
Name : ACRS
Name : TrustedPublisher
Name : REQUEST
```

To examine the specific certificates issued to the user, use the *My* certificate store. This translates to the *Personal* certificate store shown in the Certificate Manager Utility. The *Personal* certificate store is shown in Figure 16-2.

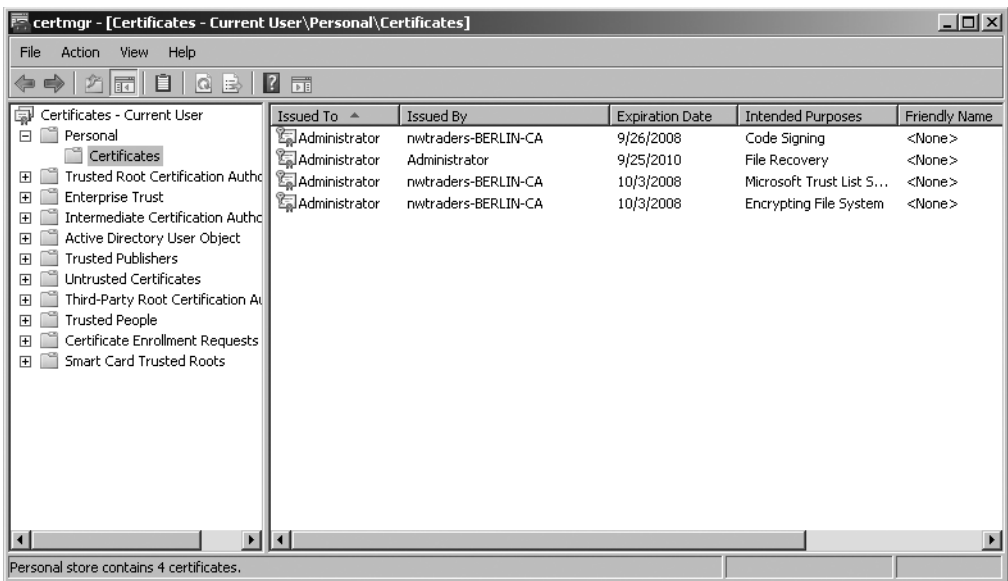


Figure 16-2 Personal certificates are stored in CurrentUser Personal certificate store.

To obtain a listing of all the personal certificates issued to the current user, use this command:

```
Get-ChildItem cert:\CurrentUser\My
```

A typical result would look something like this:

Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint	Subject
7D7F4414CCEF168ADF6BF40753B5BECDD78375931	OU=Microsoft Corporation, CN=M..
77085D8E6E5645C42DDC31771F1090D54C92FF96	CN=Administrator

A more difficult problem is trying to find a certificate that has been issued for a particular use. When a certificate is issued to a user, it's often put in the CurrentUser\My certificate store. The problem is that when using Windows PowerShell to locate the certificate, what is shown in results to the previous command lists only the thumbprint and the subject fields. In Figure 16-3 (found in the "Inspecting a Certificate" section, later in this chapter), when looking at the *Personal* certificate store (the same as CurrentUser\My), the view is quite a bit different. For example, you can readily identify which certificate is the code-signing certificate. To help solve this problem, use the FindCertificates.ps1 script. The FindCertificates.ps1 script uses the *FriendlyName* property from *EnhancedKeyUses*. These are exposed from the *System.Security.Cryptography.X509Certificates.X509ExtensionCollection* Microsoft .NET Framework class.

Begin the FindCertificates.ps1 script with a *param* statement, which is used to collect command-line arguments that control the way the script functions. There are two parameters defined. The first is the *-use* parameter, used to collect the use name of the particular certificate

in question. This can be any value related to certificate use, such as code signing, smart card user, or digital signature. Since you are doing a regular expression match, you don't need to type the entire friendly name.

The *-help* parameter is a switched parameter and doesn't need to be supplied. If it is passed to the script at run time, then the script will display the help text and exit. The *param* line of code is shown here:

```
param($use, [switch]$help)
```

Next, you must create the *funhelp()* function, used to display the help text for the script when the script is launched with the *-help* parameter specified. In the code block for the function, first create a variable named *\$helptext*, and assign the results of creating a here-string to the value of the variable. The here-string begins with the *@* symbols and ends with the *@* symbols. In between these two tags, you can ignore the Windows PowerShell quoting rules. This makes it much easier to correctly type in large amounts of text. The help text is divided into three categories: the description, the parameters, and the syntax. After completing the here-string and assigning it to the *\$helptext* variable, display the contents of the variable and exit the script. The *funhelp()* function is shown here:

```
function funHelp()
{
    $helpText=@
    DESCRIPTION:
    NAME: FindCertificates.ps1
    Finds certificates of a particular use on the local machine

    PARAMETERS:
    -use      the purpose for the certificate ex: code signing
    -help     prints help file

    SYNTAX:
    FindCertificates.ps1
    Gets a listing of all certificates in the my store

    FindCertificates.ps1 -use "digital signature"

    Gets a listing of certificates in my store that provide a digital
    signature on local computer

    FindCertificates.ps1 -use "code signing"

    Gets a listing of certificates in my store that provide code
    signing support

    FindCertificates.ps1 -help

    Prints the help topic for the script

    "@
    $helpText
    exit
}
```

You need to look for the presence of the *\$help* variable. If you find it, the script was run with the *-help* parameter specified and the user is looking for help. Print a short status message, and call the *funhelp()* function. This line of code is shown here:

```
if($help) { "Printing help now..." ; funHelp }
```

Now, you must check for the presence of the *\$use* variable. If it isn't found, then it was not supplied from the command line. Because you haven't created a default value for *\$use*, you don't have a default action for the script. Therefore, print a short status message, and call the *funhelp()* function. This line of code is shown here:

```
if(!$use) { "A use is required..." ; funHelp }
```

Next is the worker section of the script. The first step is to obtain a collection of all the certificate objects in the *My* certificate store and store them in a variable named *\$mycert* for ease of use. To do this, use the *Get-ChildItem* cmdlet, point it to the *cert:\PSDrive*, and look inside the *CurrentUser\My* certificate store. This line of code is displayed here:

```
$myCert = Get-ChildItem cert:\CurrentUser\My
```

Once you have a collection of certificate objects stored in the *\$mycert* variable, you must iterate through the collection. To do this, use the *foreach* statement with the variable *\$cert* as the enumerator. Take each certificate object individually and call the *get_extensions()* method. This returns a collection of extension objects, which are stored in the *\$certext* variable. Then you iterate through the collection of extension objects, this time using the variable *\$ext* as the enumerator. Each extension object is made up of two properties, but you are interested only in the *FriendlyName* property. Use the variable *\$name*, and once again iterate through the collection. This section of code is shown here:

```
ForEach( $cert in $myCert)
{
    $certExt = $cert.get_extensions()
    Foreach( $ext in $certExt )
    {
        foreach( $name in $ext.enhancedKeyUsages )
```

Inside the *foreach* loop, use the *if* statement and look at the *FriendlyName* property. If you find a regular expression match to the string held in the *\$use* variable (which was created from the command line), print a string that includes a header telling the user that there are matches for the string contained in the *\$use* variable. Use a subexpression to expand the *FriendlyName* value from the *\$name.friendlyname* combination. The subexpression begins with a *\$*, surrounds the *\$name.friendlyname* combination, and ends with a smooth parenthesis. Use the grave accent (line continuation character, *`*) for ease in reading, and continue the command to the next line. Use the *`n* character combination to indicate that a new line will be displayed.

Use another subexpression and expand the value of the thumbprint and the subject from the certificate object stored in the *\$cert* variable. This section of code is shown here:

```

        {
            if($name.friendlyname -match $use)
            {
                "Certificates that match $use"
                "$($name.friendlyname) certificate: `
                `n$($cert.thumbprint) `n$($cert.subject)`n"
            }
        }
    }
}

```

The completed FindCertificates.ps1 script is shown here.

FindCertificates.ps1

```
param($use, [switch]$help)
```

```
function funHelp()
```

```
{
```

```
$helpText=@"
```

```
DESCRIPTION:
```

```
NAME: FindCertificates.ps1
```

```
Finds certificates of a particular use on the local machine
```

```
PARAMETERS:
```

```
-use          the purpose for the certificate ex: code signing
```

```
-help        prints help file
```

```
SYNTAX:
```

```
FindCertificates.ps1
```

```
Gets a listing of specific certificates in the my store
```

```
FindCertificates.ps1 -use "digital signature"
```

```
Gets a listing of certificates in my store that provide a digital
signature on local computer
```

```
FindCertificates.ps1 -use "code signing"
```

```
Gets a listing of certificates in my store that provide code
signing support
```

```
FindCertificates.ps1 -help
```

```
Prints the help topic for the script
```

```
"@
```

```
$helpText
```

```
exit
```

```
}
```

```
if($help) { "Printing help now..." ; funHelp }
```

```

if(!$use) { "A use is required..." ; funHelp }

$myCert = Get-ChildItem cert:\CurrentUser\My
ForEach( $cert in $myCert)
{
    $certExt = $cert.get_extensions()
    Foreach( $ext in $certExt )
    {
        foreach( $name in $ext.enhancedKeyUsages )
        {
            if($name.friendlyname -match $use)
            {
                "Certificates that match $use"
                "$($name.friendlyname) certificate: `
                `n$($cert.thumbprint) `n$($cert.subject)`n"
            }
        }
    }
}

```

Listing Certificates

There are many times when you will want to simply list all the certificates that reside in a particular certificate store. While you can use the Windows PowerShell certificate PSDrive, you may want a little bit more control over the process. In the `ListCertificates.ps1` script, use the .NET Framework class `X509Store`. This .NET Framework class is found in the `System.Security.Cryptography.X509Certificates` namespace. Use the `New-Object` cmdlet to create an instance of this class. The `ListCertificates.ps1` script is an example of this process.

The `ListCertificates.ps1` script begins with the *param* statement; within the statement, create three parameters. The first parameter is *-store*, which is used to control which certificate store is used to provide the certificate listing. This parameter is set to default to the *My* certificate store. Next is the switched *-liststores* parameter, which causes the script to provide a complete listing of all the certificate stores on the local machine. The third parameter you'll create is the *-help* switched parameter, which is used to display help. This statement is shown here:

```
param($store="my", [switch]$listStores, [switch]$help)
```

Next is the *funhelp()* function, used to display help information. After declaring the function, begin the code block by defining a variable *\$helptext* and assigning the value of a here-string to it. In the here-string, list the description, parameters, and syntax of the script. The advantage of using a here-string is that it allows you to type in large amounts of text without typing in quotation marks. The *funhelp()* function is displayed here:

```

function funHelp()
{
    $helpText=@"
DESCRIPTION:
NAME: ListCertificates.ps1
Lists certificates on the current machine

```


PARAMETERS:

-store the certificate store to search
 -help prints help file

SYNTAX:

```
ListCertificates.ps1
```

Gets a listing of all certificates in the my store

```
ListCertificates.ps1 -store "authroot"
```

Gets a listing of certificates in authroot store on local computer

```
ListCertificates.ps1 -store "my"
```

Gets a listing of certificates in my store on local computer

```
ListCertificates.ps1 -help
```

Prints the help topic for the script

```
"@
$helpText
exit
}
```

Next is a function named *funstore()*, which provides a listing of all the certificate stores on the local machine. Begin by using the Write-Host cmdlet to print a header in green for the Current-User store location. Use the Get-ChildItem cmdlet and point it to the CurrentUser store on the cert:\PSDrive and do the same for the LocalMachine certificate location. The *funstore()* function is shown here:

```
Function funstore()
{
  write-host -foregroundcolor green "Listing currentuser stores:"
  Get-ChildItem cert:\CurrentUser
  write-host -foregroundcolor green "Listing localmachine stores:`n"
  Get-ChildItem cert:\LocalMachine
  exit
}
```

The next step is the parameter checks; that is, checking for the value of the parameter collection. First look to see if you need to display help by looking for the *\$help* variable. If you find it, call the *funhelp()* function. Look for the presence of the *\$liststore* variable. If you find this variable, call the *funstore()* function to display the available certificate stores on the computer. These two lines of code are:

```
if($help) { "Printing help now..." ; funHelp }
if($liststore) { funstore }
```

You must declare a read-only variable, using the `New-Variable` cmdlet to create a variable named `userstore`. Set the value of the variable to `currentUser`, and use the `-option` parameter to make the variable read-only. Create another variable named `$crypto`, and set that one equal to a string that represents the exact location of the `x509Store` .NET Framework class. Do this to make the code a bit easier to read, as the combination of the class name and the namespace is rather long. These two lines of code are shown here:

```
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
```

Create a new instance of the `.X509Store` .NET Framework class. Pass the path to the class along with the store location contained in the `$store` variable. Store the resulting object in the `$objstore` variable, then open the certificate store in read-only mode by using the `readonly` keyword and supplying it to the `open()` method. To produce a collection of all the certificates in the certificate store, query the `Certificates` property; store the resulting collection of certificates in the `$colcerts` variable. These three lines of code are listed here:

```
$objStore = new-object $crypto $store
$objstore.Open("ReadOnly")
$colcerts = $objstore.Certificates
```

To produce a header for the resulting listing of certificates, use the `Write-Host` cmdlet and specify the `-foreground` parameter to be blue. Print a message and use a subexpression to retrieve the number of certificates in the collection. Do this by prefacing the `ColCerts.Count` property with the `$` sign and enclosing all but the initial `$` within parentheses. This configuration looks like this: `$(($ColCerts.Count))`. This allows you to obtain the actual count of the certificates instead of expanding the object name. The code that produces the header for our output is:

```
Write-Host -ForegroundColor blue
"
    There are $(($colcerts.count)) certificates in the $store store.
    They are listed below:
"
```

Because you have obtained a collection of certificates, you must use the `foreach` statement, using the variable `$cert` as the enumerator. Use a subexpression for each of the properties you want to query for each of the certificates found in the collection. After printing the properties, close the store. This section of code is shown here:

```
foreach($cert in $colCerts)
{
    "FriendlyName: $($cert.FriendlyName)"
    "Serialnumber: $($cert.SerialNumber)"
    "Thumbprint: $($cert.thumbprint)"
    "Subject: $($cert.subject)`n"
}
$objstore.Close()
```

The completed ListCertificates.ps1 script is shown here.

ListCertificates.ps1

```
param($store="my", [switch]$listStores, [switch]$help)
```

```
function funHelp()
```

```
{
```

```
$helpText=@"
```

```
DESCRIPTION:
```

```
NAME: ListCertificates.ps1
```

```
Lists certificates on the current machine
```

```
PARAMETERS:
```

```
-store      the certificate store to search
```

```
-help       prints help file
```

```
SYNTAX:
```

```
ListCertificates.ps1
```

```
Gets a listing of all certificates in the my store
```

```
ListCertificates.ps1 -store "authroot"
```

```
Gets a listing of certificates in authroot store on  
local computer
```

```
ListCertificates.ps1 -store "my"
```

```
Gets a listing of certificates in my store on local  
computer
```

```
ListCertificates.ps1 -help
```

```
Prints the help topic for the script
```

```
"@
```

```
$helpText
```

```
exit
```

```
}
```

```
Function funstore()
```

```
{
```

```
write-host -foregroundcolor green "Listing currentuser stores:"
```

```
Get-ChildItem cert:\CurrentUser
```

```
write-host -foregroundcolor green "Listing localmachine stores:`n"
```

```
Get-ChildItem cert:\LocalMachine
```

```
exit
```

```
}
```

```
if($help) { "Printing help now..." ; funHelp }
```

```
if($liststore) { funstore }
```

```
new-variable -name userStore -value "currentUser" -option readonly
```

```
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
```

```
$objStore = new-object $crypto $store
```

```

$objstore.Open("Readonly")
$colcerts = $objstore.Certificates
Write-Host -ForegroundColor blue
"
    There are $($colcerts.count) certificates in the $store store.
    They are listed below:
"
foreach($cert in $colCerts)
{
    "FriendlyName:  $($cert.FriendlyName)"
    "SerialNumber: $($cert.SerialNumber)"
    "Thumbprint:   $($cert.thumbprint)"
    "Subject:      $($cert.subject)`n"
}
$objstore.Close()

```

Locating Expired Certificates

As certificates become more prevalent, so too does the incidence of expired certificates. Nearly everyone has connected to a Web site, perhaps to do online banking or to purchase some item from an Internet store, only to be warned that the site has an expired certificate. As a troubleshooting measure, you must be able to quickly and efficiently locate expired certificates. To do this, once again use the certificate provider for Windows PowerShell. In the FindExpiredCertificates.ps1 script, you first obtain the current date and then search the certificate store that is identified by the user from the command line.

Begin the FindExpiredCertificates.ps1 script by using the *param* statement; this script is designed to use four command-line parameters. The *-store* parameter is used to determine which certificate store will be accessed by the script. It is a required parameter, as you haven't supplied a default value and it is not a switched parameter. However, if the user does not supply a value when the script is run, then supply the *My* store as a default value. The reason you don't define the value in the *param* statement is because you want to inform the user there are other options available by pointing to the *help* switch. You will also let the user know you're using the default value for the parameter. The other *switch* statements are *-listcu*, which will list the certificate stores in the *CurrentUser* location; *-listlm*, which will list all the certificate stores in the *LocalMachine* location; and the *-help* switch, which will print the help text. The *param* statement is shown here:

```

param(
    $store,
    [switch]$listcu,
    [switch]$listlm,
    [switch]$help
)

```

Next is the *funhelp()* function, used to print the help text. To do this, begin by creating a variable, *\$helptext*, that is used to hold the help string. Use a here-string to create the help text.

Store the here-string in the *\$helptext* variable, print the contents of the variable, and exit the script. The *funhelp()* function is shown here:

```
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: FindExpiredCertificates.ps1
Finds expired certificates on the local machine

PARAMETERS:
-store      the certificate store on the computer
-help       prints help file

SYNTAX:
FindExpiredCertificates.ps1
Gets a listing of expired certificates in the my store of the
currentuser

FindExpiredCertificates.ps1 -store "currentuser\my"

Gets a listing of expired certificates in the my store of the
currentuser

FindExpiredCertificates.ps1 -store "currentuser\smartcardroot"

Gets a listing of expired certificates in the smartcardroot store
of the currentuser

FindExpiredCertificates.ps1 -listcu

Gets a listing of certificate stores for the
currentuser

FindExpiredCertificates.ps1 -listlm

Gets a listing of certificate stores for the
localmachine

FindExpiredCertificates.ps1 -help

Prints the help topic for the script

"@
$helpText
exit
}
```

You must parse the command line and see what parameters have been supplied. The first parameter to check for is the *-help* switch; if you find the *\$help* variable, then the script was run with the *-help* switch. Print a short message, and call the *funhelp()* function. This line of code is shown here:

```
if($help) { "Printing help now..." ; funHelp }
```

Look for the *-listcu* switch; if you find the *\$listcu* variable, the script was launched with the *-listcu* switch. Print a short status message, and use the *Get-ChildItem* cmdlet to produce a listing of certificate stores in the *CurrentUser* location. Once this has been done, exit the script. This section of code is shown here:

```
if($listcu) {
    "Certificate stores in currentuser"
    get-childitem cert:\currentuser ; exit
}
```

The *-listlm* switch is the next step. If you find the *\$listlm* variable, the script was launched with the *-listlm* switch. Print a status message, and use the *get-ChildItem* cmdlet to produce a listing of certificate stores in the *LocalMachine* location. After this is completed, exit the script. This section of code is shown here:

```
if($listlm) {
    "Certificate stores in localmachine"
    get-childitem cert:\localmachine ; exit
}
```

The next parameter, *-store*, is used to control which certificate store will be searched for expired certificates. If the *-store* switch is not used; you will default to looking in the *CurrentUser\My* store. Print a message that tells the user that you are using defaults, use the *\$myinvocation.mycommand* command to print the name of the script that is run, and suggest using the *-help* switch to view additional examples. This line of code is shown here:

```
if(!$store) {
    $store = "currentuser\my"
    "Using default store: $store"
    "See $($myinvocation.mycommand) -help" `
    + " for additional examples"
}
```

The *FindExpiredCertificates.ps1* script provides coding to print the message for using the default certificate store. Because the goal is for the output to be on a single line, close the quotation marks and use the grave accent (line continuation or ```) on the first line. Concatenate the second line of text by using `+` for the remainder of the string. If you just continue the string to the next line without closing the quotation marks, you'll end up with two lines printed in the console.

Next is the reference section of the script.



More Info The four parts of a script are detailed in my book *Microsoft VBScript Step by Step* (Microsoft Press, 2006). Even though the book is about VBScript, it is an excellent primer on scripting in general, and most of the same principles apply.

Obtain a *datetime* object by using the *Get-Date* cmdlet, and store the object in the *\$currentdate* variable. The second step is to use the *Get-ChildItem* cmdlet to retrieve a collection of all the

certificates in the store pointed to by the value in the *\$store* variable. The *\$colcert* variable is used to contain the collection of certificates. These two lines of code are shown here:

```
$currentDate = Get-Date
$colcert = Get-ChildItem cert:\$store
```

Now use the Write-Host cmdlet and specify the *-foregroundcolor* parameter to print a message in cyan. Use the *foreach* statement to iterate through the collection of certificates, using the variable *\$cert* as the enumerator. Once you have an individual certificate stored in the *\$cert* variable, examine the *NotAfter* property to see if it is less than the value stored in the *\$current-date* variable. If it is, then print both the thumbprint and the date in which the certificate expired. This section of code is shown here:

```
Write-host -foregroundcolor cyan "Expired Certificates in $store"
foreach($cert in $colcert)
{
    if($cert.notafter -lt $currentDate)
    {
        Write-host `
            "
                $($cert.thumbprint) `t $($cert.NotAfter)
            "
    }
}
```

The complete FindExpiredCertificates.ps1 script can be examined here.

FindExpiredCertificates.ps1

```
param(
    $store,
    [switch]$listcu,
    [switch]$listlm,
    [switch]$help
)

function funHelp()
{
    $helpText=@"
DESCRIPTION:
NAME: FindExpiredCertificates.ps1
Finds expired certificates on the local machine

PARAMETERS:
-store      the certificate store on the computer
-help      prints help file

SYNTAX:
FindExpiredCertificates.ps1
Gets a listing of expired certificates in the my store of the
currentuser

FindExpiredCertificates.ps1 -store "currentuser\my"
```


Gets a listing of expired certificates in the my store of the currentuser

```
FindExpiredCertificates.ps1 -store "currentuser\smartcardroot"
```

Gets a listing of expired certificates in the smartcardroot store of the currentuser

```
FindExpiredCertificates.ps1 -listcu
```

Gets a listing of certificate stores for the currentuser

```
FindExpiredCertificates.ps1 -listlm
```

Gets a listing of certificate stores for the localmachine

```
FindExpiredCertificates.ps1 -help
```

Prints the help topic for the script

```
"@
$helpText
exit
}

if($help) { "Printing help now..." ; funHelp }
if($listcu) {
    "Certificate stores in currentuser"
    get-childitem cert:\currentuser ; exit
}
if($listlm) {
    "Certificate stores in localmachine"
    get-childitem cert:\localmachine ; exit
}
if(!$store) {
    $store = "currentuser\my"
    "Using default store: $store"
    "See $($myinvocation.mycommand) -help" `
    + " for additional examples"
}

$currentDate = Get-Date
$colcert = Get-ChildItem cert:\$store
Write-host -foregroundcolor cyan "Expired Certificates in $store"
foreach($cert in $colcert)
{
    if($cert.notafter -lt $currentDate)
    {
        Write-host `
        "
        $($cert.thumbprint) `t $($cert.Notafter)
        "
    }
}
}
```

Identifying Certificates about to Expire

When you issue certificates to users, you'll eventually run into a problem. That's because, in general, many user certificates are only good for only one or two years. This means that there will nearly always be users who need to sign an e-mail, use a laptop, make a remote connection to the network, sign some code, or encrypt a file; they may not be able to take these actions because of an expired certificate.

That's why proactive scripting has great potential. Using the `FindCertificatesAboutToExpire.ps1` script, you can examine certificate expiration dates to see which will expire on or before a future date.

In the `FindCertificatesAboutToExpire.ps1` script, begin with the *param* statement and create five parameters. One parameter is required, one has a default value, and the other three are switched. The *-store* parameter is the required one, and just like the `FindExpiredCertificates.ps1` script, you must check for the presence of the *\$store* variable and supply a value if it is missing. The *-days* parameter is set to a default value of 30 days. The *-listcu* parameter is used to list available certificate stores in the `CurrentUser` location. The *-listlm* parameter produces a similar listing for the `LocalMachine` location. The *-help* parameter prints out help. The *param* statement is shown here:

```
param(
    $store,
    $days=30,
    [switch]$listcu,
    [switch]$listlm,
    [switch]$help
)
```

Next is the *funhelp()* function, which prints help for the script, including several samples of the syntax. The *funhelp()* function first creates a *\$helptext* variable to store the help text message. To produce the help text, use a here-string, which allows you to avoid quoting issues. Create the description, parameters, and syntax section of the help text, then print the contents of the *\$helptext* variable, and exit the script. The *funhelp()* function is shown here:

```
function funHelp()
{
    $helpText=@"
DESCRIPTION:
NAME: FindCertificatesAboutToExpire.ps1
Finds certificates about to expire with in a certain
number of days on the local machine

PARAMETERS:
-store      the certificate store on the computer
-days       number of days in the future to evaluate for
             certificate expiration
-help       prints help file

SYNTAX:
FindCertificatesAboutToExpire.ps1
```

Gets a listing of certificates about to expire within 30 days
in the my store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -days 45
```

Gets a listing of certificates about to expire within 45 days
in the my store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -store "currentuser\my" -days 60
```

Gets a listing of certificates about to expire within 60 days
in the my store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -store "currentuser\smartcardroot"
```

Gets a listing of certificates about to expire within 30 days
in the smartcardroot store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -listcu
```

Gets a listing of certificate stores for the
currentuser

```
FindCertificatesAboutToExpire.ps1 -listlm
```

Gets a listing of certificate stores for the
localmachine

```
FindCertificatesAboutToExpire.ps1 -help
```

Prints the help topic for the script

```
"@
$helpText
exit
}
```

After completing the *funhelp()* function, work on executing the first code. You'll need to check the parameters: The first one is the *-help* parameter; if it's present, it allows you to call the *funhelp()* function and execute the script. This line of code is shown here:

```
if($help) { "Printing help now..." ; funHelp }
```

If you find the *\$listcu* variable, print a status message, use the *Get-ChildItem* cmdlet to produce a list of all the certificate stores in the *CurrentUser* location, and exit the script. Perform a similar series of steps for the *-listlm* parameter: If you find the *\$listlm* variable, print a status message, call the *Get-ChildItem* cmdlet to produce a list of all the certificate stores in the *LocalMachine* location, then exit the script. This section of code is shown here:

```
if($listcu) {
    "Certificate stores in currentuser"
    get-childitem cert:\currentuser ; exit
}
```

```
if($listlm) {
    "Certificate stores in localmachine"
    get-childitem cert:\localmachine ; exit
}
```

If the required parameter, *-store*, is not supplied, then the *\$store* variable will be absent. If you detect this condition, use the *My* store for the query. However, you'll also want to inform the user that there are other options available. To do this, let the user know that you're using a default value, and use the *\$myinvocation.mycommand* command to print the script name. To obtain the script name, you must use a subexpression. Suggestion: Use help to see examples. This section of code is shown here:

```
if(!$store) {
    $store = "currentuser\my"
    "Using default store: $store"
    "See $($myinvocation.mycommand) -help" `
    + " for additional examples"
}
```

After checking the parameters, start on the worker portion of the script, first creating an instance of the *System.DateTime* .NET Framework object and using the *adddays()* method to add days to the current date. Store the future date in the variable named *\$currentdate*. Obtain a collection of certificates by using the *Get-ChildItem* cmdlet. These two lines of code are shown here:

```
$currentDate = (Get-Date).adddays($days)
$colcert = Get-ChildItem cert:\$store
```

Print a header for the output. To do this, use the *Write-Host* cmdlet as shown here:

```
Write-host -foregroundcolor cyan "Certificates in $store that" `
    " expire in $days days"
```

Use the *foreach* statement to iterate through the collection of certificates, using the *\$cert* variable as an enumerator to keep track of your location in the collection. Examine the *NotAfter* property of each certificate, which is the expiration date. If the date is less than the future date stored in the *\$currentdate* variable, print the thumbprint and the expiration date. This section of code is shown here:

```
foreach($cert in $colcert)
{
    if($cert.notafter -lt $currentDate)
    {
        Write-host `
            "
            $($cert.thumbprint) `t $($cert.NotAfter)
            "
    }
}
```

The completed FindCertificatesAboutToExpire.ps1 script is shown here.

FindCertificatesAboutToExpire.ps1

```
param(
    $store,
    $days=30,
    [switch]$listcu,
    [switch]$listlm,
    [switch]$help
)
function funHelp()
{
    $helpText=@"
DESCRIPTION:
NAME: FindCertificatesAboutToExpire.ps1
Finds certificates about to expire with in a certain
number of days on the local machine

PARAMETERS:
-store      the certificate store on the computer
-days      number of days in the future to evaluate for
            certificate expiration
-help      prints help file
```

SYNTAX:

```
FindCertificatesAboutToExpire.ps1
```

Gets a listing of certificates about to expire within 30 days
in the my store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -days 45
```

Gets a listing of certificates about to expire within 45 days
in the my store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -store "currentuser\my" -days 60
```

Gets a listing of certificates about to expire within 60 days
in the my store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -store "currentuser\smartcardroot"
```

Gets a listing of certificates about to expire within 30 days
in the smartcardroot store of the currentuser

```
FindCertificatesAboutToExpire.ps1 -listcu
```

Gets a listing of certificate stores for the
currentuser

```
FindCertificatesAboutToExpire.ps1 -listlm
```

Gets a listing of certificate stores for the
localmachine

```
FindCertificatesAboutToExpire.ps1 -help
```

Prints the help topic for the script

```
"@
$helpText
exit
}

if($help) { "Printing help now..." ; funHelp }
if($listcu) {
    "Certificate stores in currentuser"
    get-childitem cert:\currentuser ; exit
}
if($listlm) {
    "Certificate stores in localmachine"
    get-childitem cert:\localmachine ; exit
}
if(!$store) {
    $store = "currentuser\my"
    "Using default store: $store"
    "See $($myinvocation.mycommand) -help" `
    + " for additional examples"
}

$currentDate = (Get-Date).adddays($days)
$colcert = Get-ChildItem cert:\$store
Write-host -foregroundcolor cyan "Certificates in $store that" `
    " expire in $days days"

foreach($cert in $colcert)
{
    if($cert.notafter -lt $currentDate)
    {
        Write-host `
            "
            $($cert.thumbprint) `t $($cert.Notafter)
            "
    }
}
```

Managing Certificates

There are several tasks that fall under the purview of certificate management, including importing certificates, inspecting certificates, and deleting certificates. This section examines each of these tasks.

Inspecting a Certificate

Before importing a certificate, you may want to inspect it to ensure it is the correct certificate for the operation at hand. The Certificate Manager utility that has been used in the past does not have this capability. To inspect a certificate, use the .NET Framework class *X509Certificate*. The *X509Certificate* class is located in the *Security.Cryptography.X509Certificates* .NET Framework

namespace. The properties to inspect are the same properties shown in the Certificate Manager utility, Figure 16-3. These same properties are examined in the `InspectCertificate.ps1` script.

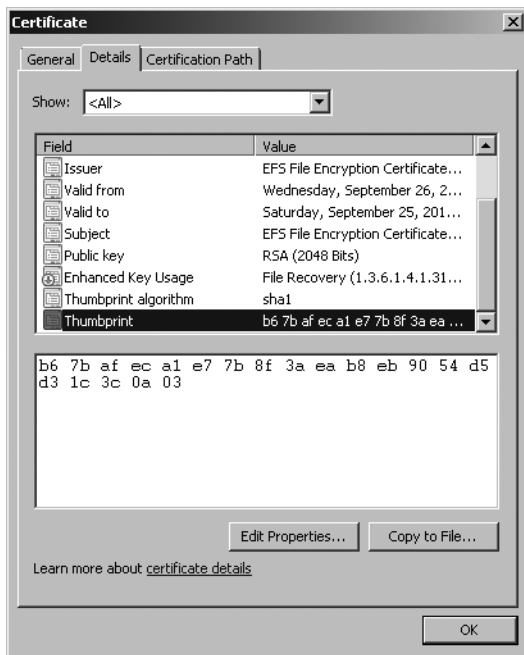


Figure 16-3 Certificate properties as observed in the Certificate Manager utility.

Begin the `InspectCertificate.ps1` script with the *param* statement. There are two necessary parameters for this script. The first one is the *-cert* parameter, which is used to include the full path and name of the certificate to inspect. The second one is the *-help* parameter, which is a switched parameter to display help if requested. The *param* statement is shown here:

```
param($cert, [switch]$help)
```

Next is the *funhelp()* function, which is used to display help information when the script is launched with the *-help* parameter. First use the *function* statement to create the *funhelp()* function. Begin the code block for the function by using braces (`{ }`). Inside the code block, create a variable, *\$helptext*, and assign the results of a here-string to it. The here-string is created by using the `@` and `@` at the beginning and end of the string. Inside the here-string, you don't need quotation marks; this feature simplifies the creation of large text blocks. After the here-string is created, print the contents of the *\$helptext* variable, and exit the script by using the *exit* statement. The *funhelp()* function is shown here:

```
function funHelp()
{
$helpText="@
DESCRIPTION:
NAME: InspectCertificate.ps1
```

Finds certificates of a particular use on the local machine

PARAMETERS:

-cert the full path to the certificate to inspect
-help prints help file

SYNTAX:

InspectCertificate.ps1

Generates an error that a certificate is required

```
InspectCertificate.ps1 -cert "c:\fso\filerecovery.cer"
```

Inspects a certificate called filerecovery in the c:\fso directory. This certificate could be DER encoded or base -64 encoded .cer file.

```
InspectCertificate.ps1 -help
```

Prints the help topic for the script

```
"@
$helpText
exit
}
```

You now must look for command-line parameters. The first parameter checks for the presence of the *\$help* variable by using the *if* statement. In the code block for the *if* statement, print a string, and call the *funhelp()* function. Placing a semicolon on the line allows you to place two unrelated commands on the same line. Check for the absence of the *\$cert* variable by using the *not* operator (!). If the *\$cert* variable is not found, call the *funhelp()* function as well. These two lines of code are shown here:

```
if($help) { "Printing help now..." ; funHelp }
if(!$cert) { "A certificate is required..." ; funHelp }
```

Now, you must make the connection to the certificate, using the *X509Certificate* .NET Framework class, which is in the *Security.Cryptography.X509Certificates* .NET Framework namespace.

Working with .NET Framework Classes

In the *InspectCertificate.ps1* script, you'll use a shortcut method for creating an instance of the *X509Certificate* class. You use the same method when you do typecasting. As an example, if you want to create a string, you can *cast* it to the *System.String* type by using the following syntax:

```
[string]"This is a string"
```

If you want to create an instance of the *X509Certificate*, use the same syntax as shown here:

```
$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"
```


To understand this in a bit more detail, examine the `ThreeStrings.ps1` script, following. First use the variable `$a` to hold a string in the usual fashion—by assigning the string to the variable. Use the `gettype()` method to prove this is a `System.String` .NET Framework class. Now use a short name, `[string]`, to once again create a string. This time, assign it to the variable `$b`, which also reports that the type of object is a `System.String`. Finally, use the full name `[system.string]` within brackets (`[]`) and assign the result to the `$c` variable, which once again reports `System.String`. The `ThreeStrings.ps1` script is shown here.

ThreeStrings.ps1

```
$a = "`$a is a string"
$a
"$a : It is a $($a.gettype())`n"

$b = [string]"`$b is a string"
$b
"$b : It is a $($b.gettype())`n"

$c = [system.string]"`$c is a string"
$c
"$c : It is a $($c.gettype())`n"

"A $($c.gettype()) .NET framework class has the " `
+ "members"
$a | get-member
```

To connect to the certificate, use the `$cert` variable containing the certificate object and point to the .NET Framework `X509Certificate` class. Store the new object in the `$objcert` variable. This line of code is shown here:

```
$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"
```

The remainder of the script uses subexpressions to print the results of several method calls. This is the first instance shown of using a subexpression to return the result of a method in a text string. The last two items in the output section of the script are properties: *Issuer* and *Subject*. The entire output section of the script is shown here:

```
"HashString: $($objCert.GetCertHashString())"
"EffectiveDate: $($objCert.GetEffectiveDateString())"
"ExpirationDate: $($objCert.GetExpirationDateString())"
"HashCode: $($objCert.GetHashCode())"
"KeyAlgorithm: $($objCert.GetKeyAlgorithm())"
"KeyAlgorithmParameters: $($objCert.GetKeyAlgorithmParametersString())"
"Name: $($objCert.GetName())`n"
"PublicKey: $($objCert.GetPublicKeyString())`n"
"RawCertData: $($objCert.GetRawCertDataString())`n"
"SerialNumber: $($objCert.GetSerialNumberString())"
"Cert: $($objCert.ToString())"
"Issuer: $($objCert.Issuer)"
"Subject: $($objCert.Subject)"
```

The completed InspectCertificate.ps1 script is shown here.

InspectCertificate.ps1

```
param($cert, [switch]$help)
```

```
function funHelp()
```

```
{
```

```
$helpText=@"
```

```
DESCRIPTION:
```

```
NAME: InspectCertificate.ps1
```

```
Finds certificates of a particular use on the local machine
```

```
PARAMETERS:
```

```
-cert      the full path to the certificate to inspect
```

```
-help      prints help file
```

```
SYNTAX:
```

```
InspectCertificate.ps1
```

```
Generates an error that a certificate is required
```

```
InspectCertificate.ps1 -cert "c:\fso\filerecovery.cer"
```

Inspects a certificate called filerecovery in the c:\fso directory. This certificate could be DER encoded or base -64 encoded .cer file.

```
InspectCertificate.ps1 -help
```

Prints the help topic for the script

```
"@
```

```
$helpText
```

```
exit
```

```
}
```

```
if($help) { "Printing help now..." ; funHelp }
```

```
if(!$cert) { "A certificate is required..." ; funHelp }
```

```
$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"
```

```
"HashString: $($objCert.GetCertHashString())"
```

```
"EffectiveDate: $($objCert.GetEffectiveDateString())"
```

```
"ExpirationDate: $($objCert.GetExpirationDateString())"
```

```
"HashCode: $($objCert.GetHashCode())"
```

```
"KeyAlgorithm: $($objCert.GetKeyAlgorithm())"
```

```
"KeyAlgorithmParameters: $($objCert.GetKeyAlgorithmParametersString())"
```

```
"Name: $($objCert.GetName())`n"
```

```
"PublicKey: $($objCert.GetPublicKeyString())`n"
```

```
"RawCertData: $($objCert.GetRawCertDataString())`n"
```

```
"SerialNumber: $($objCert.GetSerialNumberString())"
```

```
"Cert: $($objCert.ToString())"
```

```
"Issuer: $($objCert.Issuer)"
```

```
"Subject: $($objCert.Subject)"
```

Importing a Certificate

After receiving a new certificate, you may want to import it into your certificate store. In Figure 16-4, you can see the Certificate Import Wizard. You also can import a certificate by using a Windows PowerShell script; there's an example of this in the `ImportCertificate.ps1` script.



Figure 16-4 The Certificate Import Wizard defaults to importing certificates to the Personal certificate store.

Begin the `ImportCertificate.ps1` script with the *param* statement, which defines four parameters. The first one, *-cert*, holds the path to the certificate to import. The *-store* parameter defaults to the *My* store, which Certificate Manager refers to as the *Personal* certificate store. There are two switches. The first one, *-liststores*, lists available certificate stores in the *currentuser* namespace. The last switch, *-help*, displays help. The *param* portion of the script is shown here:

```
param(  
    $cert,  
    $store = "my",  
    [switch]$liststores,  
    [switch]$help  
)
```

Next is the *funhelp()* function, which displays help for the script when it is called from the command line by the *-help* switch. The *funhelp()* function consists of three sections inside a giant here-string. The first section is the description, the second includes the parameters, and the third section is the syntax. The results of the here-string are stored in the *\$helptext* variable

and are displayed at the end of the function. The *funhelp()* function then exits the script. This function is shown here:

```
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: ImportCertificate.ps1
Imports a certificate into a certificate store

PARAMETERS:
-cert          path of certificate to import
-store         the certificate store on the computer
-liststores    lists certificate stores on local machine
-help         prints help file

SYNTAX:
ImportCertificate.ps1

Prints error message a certificate is required, and displays
help

ImportCertificate.ps1 -cert "c:\fso\mycert.pfx"

Imports a certificate stored in the c:\fso folder named
mycert.pfx into the my store of the currentuser

ImportCertificate.ps1 -store "my" -cert
"c:\fso\mycert.pfx"

Imports a certificate stored in the c:\fso folder named
mycert.pfx into the my store of the currentuser

ImportCertificate.ps1 -store "smartcardroot"
-cert "c:\fso\mycert.pfx"

Imports a certificate stored in the c:\fso folder named
mycert.pfx into the smartcardroot store of the currentuser

ImportCertificate.ps1 -liststores

Gets a listing of certificate stores for the
currentuser

ImportCertificate.ps1 -help

Prints the help topic for the script

"@
$helpText
exit
}
```

Next is the *funstore()* function, used to display all the certificate stores on the current machine. It begins with the CurrentUser store and ends with the LocalMachine store. To produce the

list, use the `Get-ChildItem` cmdlet. To print the header for each listing, use the `Write-Host` cmdlet. The `funstore()` function is shown here:

```
Function funstore()
{
    write-host -foregroundcolor green "Listing currentuser stores:"
    Get-ChildItem cert:\CurrentUser
    write-host -foregroundcolor green "Listing localmachine stores:`n"
    Get-ChildItem cert:\LocalMachine
    exit
}
```

Check the command-line parameters. The first parameter to look for is the `-help` parameter. Do this by checking for the existence of the `$help` variable. If you find the `$help` variable, print a message, and call the `funhelp()` function. Look for the `-liststores` switch. If you find the `$liststores` variable, call the `funstore()` function. Finally, check for the absence of the `$cert` variable. If the `-cert` switch is not used and if you didn't specify either the `-help` or the `-liststores` switches, print an error message that a certificate is required, call the `funhelp()` function, and exit the script. These three checks are shown here:

```
if($help) { "Printing help now..." ; funHelp }
if($liststores) { funStore }
if(!$cert) {
    "A certificate path is required..." ;
    funhelp
}
```

You must declare two variables. The first one to create is the `$userstore` variable. Set it to a value of `CurrentUser`, and mark it as read-only. The second variable to create is `$crypto`. Assign it the value of a string to represent the .NET Framework class you want to work with. These two variable assignments are shown here:

```
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
```

You must create an instance of the `X509Store` class. To do this, use the `New-Object` cmdlet and pass it the string held in the `$crypto` variable, along with the store location contained in the `$store` variable. The last parameter the `X509Store` class requires is the specific certificate store, which is contained in the `$userstore` variable. Hold the returned `X509Store` object in the `$objstore` variable. This line of code is shown here:

```
$objStore = new-object $crypto $store, $userStore
```

After creating the `X509Store` object, use the `open()` method and select the `readwrite` mode of operation. Call the `add()` method and pass it the `$cert` variable, which contains an instance of the certificate object. Finally, call the `close()` method. This section of code is shown here:

```
$objstore.Open("ReadWrite")
$objstore.Add($cert)
$objstore.Close()
```

The completed ImportCertificate.ps1 script is shown here.

ImportCertificate.ps1

```
param(
    $cert,
    $store = "my",
    [switch]$liststores,
    [switch]$help
)
```

```
function funHelp()
{
    $helpText=@'
DESCRIPTION:
NAME: ImportCertificate.ps1
Imports a certificate into a certificate store
```

```
PARAMETERS:
-cert      path of certificate to import
-store     the certificate store on the computer
-liststores lists certificate stores on local machine
-help      prints help file
```

```
SYNTAX:
ImportCertificate.ps1
```

```
Prints error message a certificate is required, and displays
help
```

```
ImportCertificate.ps1 -cert "c:\fso\mycert.pfx"
```

```
Imports a certificate stored in the c:\fso folder named
mycert.pfx into the my store of the currentuser
```

```
ImportCertificate.ps1 -store "my" -cert
"c:\fso\mycert.pfx"
```

```
Imports a certificate stored in the c:\fso folder named
mycert.pfx into the my store of the currentuser
```

```
ImportCertificate.ps1 -store "smartcardroot"
-cert "c:\fso\mycert.pfx"
```

```
Imports a certificate stored in the c:\fso folder named
mycert.pfx into the smartcardroot store of the currentuser
```

```
ImportCertificate.ps1 -liststores
```

```
Gets a listing of certificate stores for the
currentuser
```

```
ImportCertificate.ps1 -help
```

```
Prints the help topic for the script
```

```

"@
    $helpText
    exit
}

Function funstore()
{
    write-host -foregroundcolor green "Listing currentuser stores:"
    Get-ChildItem cert:\CurrentUser
    write-host -foregroundcolor green "Listing localmachine stores:`n"
    Get-ChildItem cert:\LocalMachine
    exit
}

if($help) { "Printing help now..." ; funHelp }
if($liststores) { funStore }
if(!$cert) {
    "A certificate path is required..." ;
    funhelp
}
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store, $userStore
$objStore.Open("ReadWrite")
$objStore.Add($cert)
$objStore.Close()

```

Deleting a Certificate

There are times when you will want to remove a certificate from the certificate store. This is common when a certificate has expired, if you no longer trust the issuer of the certificate, or if the certificate chain is broken. If you have only a few certificates to delete, you can easily use the Certificate Manager utility. However, if you have many certificates, you'll want to script the removal of the offending certificates. To do this, use the `DeleteCertificates.ps1` script.

To use the `DeleteCertificate.ps1` script, begin with the *param* statement and four parameters. The first parameter, *-cert*, is required. The second parameter, *-store*, is set to a default value of the *My* store. Next come two switched parameters. The first one, *-listcerts*, causes the script to print a listing of all the scripts in the selected certificate store. The last parameter is the *-help* parameter, which prints a list of all the certificates in the select store. The *param* statement is shown here:

```

param(
    $cert,
    $store = "my",
    [switch]$listcerts,
    [switch]$help
)

```

Next is the *funhelp()* function, which prints a help message when the script is run with the *-help* parameter. To do this, create a variable named *\$helptext* and assign the results of a

here-string to it. In the here-string, create three sections: The first is the description section that describes the purpose of the script. The second is the parameters section listing the parameters supported by the script. The last is the syntax section that describes the syntax of the various parameters. After the here-string is created, assign it to the *\$helptext* variable, display the contents of the variable, and exit the script. The *funhelp()* function is shown here:

```
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: DeleteCertificate.ps1
Removes a certificate from a certificate store

PARAMETERS:
-store      the certificate store on the computer
-cert       certificate to delete
-listcerts  lists certificates in specified store
-help       prints help file

SYNTAX:
DeleteCertificate.ps1

Prints error message a certificate is required, and displays
help

DeleteCertificate.ps1 -cert "B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03"

Removes a certificate with thumbprint of
B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03 from the my store of
the currentuser

DeleteCertificate.ps1 -store "my" -cert "OU=EFS File Encryption
Certificate"

Removes a certificate with subject of
OU=EFS File Encryption Certificate from the my store
of the currentuser

DeleteCertificate.ps1 -store "smartcardroot"
-cert "E47F375796238DB54CB70DA7A5E88F79"

Removes a certificate with the serial number of
E47F375796238DB54CB70DA7A5E88F79 from the smartcardroot
store of the currentuser

DeleteCertificate.ps1 -listcerts

Gets a listing of certificates for the my store of the
currentuser

DeleteCertificate.ps1 -help

Prints the help topic for the script
```



```
"@
$helpText
exit
}
```

Create the *funcert()* function by first creating a variable, *\$crypto*, that holds a string representing the namespace and class name of the *X509Store* .NET Framework class. Do this only for readability as the combination of the namespace and the class name is rather long. Use the *New-Object* cmdlet to create a new instance of the *X509Store* class. The constructor for this class needs both a store location and the name of a certificate store within that location. Use the values contained in the *\$store* variable and the *\$userstore* variable, and hold the returned *X509Store* object in the *\$objstore* variable.

Next, use the *open()* method to open the certificate store; open the store in *readwrite* mode to allow access to the certificates within the store. Create a collection of certificates by querying the *Certificates* property and hold the collection of certificates in the *\$colcerts* variable. Print a header to the list of certificates by using the *Write-Host* cmdlet. This section of code is shown here:

```
Function funcert()
{
    $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
    $objStore = new-object $crypto $store, $userStore
    $objstore.Open("ReadWrite")
    $colcerts = $objstore.Certificates
    Write-Host -ForegroundColor blue
    "
    There are $($colcerts.count) certificates in the $store store.
    They are listed below:
    "
```

After printing the list header, iterate through the collection of certificates contained in the *\$colcerts* variable. Use the variable *\$cert* as the enumerator to keep track of the individual certificates as you move through the collection. After storing an individual certificate in the *\$cert* variable, print its *Friendlyname*, *Serialnumber*, *Thumbprint*, and *Subject* to the screen. After making your way through the collection, close the store, and exit the script. This section of code is shown here:

```
foreach($cert in $colCerts)
{
    "FriendlyName: $($cert.FriendlyName)"
    "Serialnumber: $($cert.SerialNumber)"
    "Thumbprint: $($cert.thumbprint)"
    "Subject: $($cert.subject)`n"
}
$objjstore.Close()
exit
}
```

Next is the *findcert()* function, where you search for a specific certificate. If you find the certificate, store the returned certificate object in the global variable *\$mycert*. Begin the *findcert()* function by creating a variable, *\$crypto*, to hold the string representing the .NET Framework class *X509Store* and its associated namespace, *System.Security.Cryptography.X509Certificates*. Use the *New-Object* cmdlet to create an instance of the *X509Store* object. To do this, provide it the string representing the class path, the variable containing the name of the certificate store location, and the variable containing the name of the certificate store. Store the returned *X509Store* object in the *\$objstore* variable. After creating the *X509Store* object, use the *open()* method to open the certificate store. Supply the keyword *readwrite* to allow modification of the certificate store. Query the *Certificates* property, which returns a collection of certificates in the store. This section of code is shown here:

```
Function findcert($key)
{
    $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
    $objStore = new-object $crypto $store, $userStore
    $objstore.Open("ReadWrite")
    $colcerts = $objstore.Certificates
```

Having obtained a collection of certificates, use the *foreach* cmdlet to iterate through the collection contained in the *\$colcerts* variable. Use the *\$cert* variable as the enumerator as you go through the collection. When there is a single variable contained in the *\$cert* variable, query the *Thumbprint*, *SerialNumber*, *FriendlyName*, and *Subject* properties of the certificate object to see if you can match the value contained in the *\$key* variable. After finding a match, store the certificate object in the global variable *\$mycert*. This section of code is shown here:

```
foreach($cert in $colCerts)
{
    if($cert.thumbprint -match $key) { $global:mycert = $cert }
    if($cert.serialnumber -match $key) { $global:mycert = $cert }
    if($cert.friendlyname -match $key) { $global:mycert = $cert }
    if($cert.subject -match $key) { $global:mycert = $cert }
}
}
```

After creating all these functions, you finally get to the first lines of code executed when the script is run. First create a read-only variable named *\$userstore* and assign the value *CurrentUser* to it. Next, initialize the *\$mycert* variable as a global variable and assign the value of null to it. These two lines of code are shown here:

```
new-variable -name userStore -value "currentUser" -option readonly
$global:mycert = $null
```

It is now time to check the command-line parameters. The first one to check is the *-help* parameter. If you find the *\$help* variable, print a message string, and call the *funhelp()* function. If you find the *\$listcerts* variable, call the *funcert()* function. Finally, check for the presence

of the *\$cert* variable. If you don't find it, print an error message and call the *funhelp()* function. These three parameter checks are shown here:

```
if($help) { "Printing help now..." ; funHelp }
if($listcerts) { "Listing certificates in $store" ; funcert }
if(!$cert) {
    "A certificate is required..." ;
    funhelp
}
```

When you're certain that the command-line parameters are satisfactory, call the *findcert()* function and pass it the certificate name contained in the *\$cert* variable. After retrieving the certificate object and storing it in the *\$mycert* variable, create an instance of the *X509Store* .NET Framework class, open the certificate store, and call the *remove()* method while passing it the certificate object contained in the *\$mycert* variable. After this, close the certificate store. This section of code is shown here:

```
Findcert($cert)

$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objstore = new-object $crypto $store, $userstore
$objstore.Open("ReadWrite")
$objstore.Remove($mycert)
$objstore.Close()
```

The completed *DeleteCertificate.ps1* script is shown here.

DeleteCertificate.ps1

```
param(
    $cert,
    $store = "my",
    [switch]$listcerts,
    [switch]$help
)

function funHelp()
{
    $helpText=@"
DESCRIPTION:
NAME: DeleteCertificate.ps1
Removes a certificate from a certificate store

PARAMETERS:
-store      the certificate store on the computer
-cert       certificate to delete
-listcerts  lists certificates in specified store
-help       prints help file

SYNTAX:
DeleteCertificate.ps1
```

```
Prints error message a certificate is required, and displays
help
```

```
DeleteCertificate.ps1 -cert "B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03"
```

Removes a certificate with thumbprint of
B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03 from the my store of
the currentuser

```
DeleteCertificate.ps1 -store "my" -cert "OU=EFS File Encryption  
Certificate"
```

Removes a certificate with subject of
OU=EFS File Encryption Certificate from the my store
of the currentuser

```
DeleteCertificate.ps1 -store "smartcardroot"  
-cert "E47F375796238DB54CB70DA7A5E88F79"
```

Removes a certificate with the serial number of
E47F375796238DB54CB70DA7A5E88F79 from the smartcardroot
store of the currentuser

```
DeleteCertificate.ps1 -listcerts
```

Gets a listing of certificates for the my store of the
currentuser

```
DeleteCertificate.ps1 -help
```

Prints the help topic for the script

```
"@  
$helpText  
exit  
}
```

```
Function funcert()  
{  
    $crypto = "System.Security.Cryptography.X509Certificates.X509Store"  
    $objStore = new-object $crypto $store, $userStore  
    $objstore.Open("ReadWrite")  
    $colcerts = $objstore.Certificates  
    Write-Host -ForegroundColor blue  
    "  
    There are $($colcerts.count) certificates in the $store store.  
    They are listed below:  
    "  
    foreach($cert in $colCerts)  
    {  
        "FriendlyName: $($cert.FriendlyName)"  
        "Serialnumber: $($cert.SerialNumber)"  
        "Thumbprint: $($cert.thumbprint)"  
        "Subject: $($cert.subject)`n"  
    }  
    $objstore.Close()  
    exit
```

```

}

Function findcert($key)
{
    $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
    $objStore = new-object $crypto $store, $userStore
    $objstore.Open("ReadWrite")
    $colcerts = $objstore.Certificates

    foreach($cert in $colCerts)
    {
        if($cert.thumbprint -match $key) { $global:mycert = $cert }
        if($cert.serialnumber -match $key) { $global:mycert = $cert }
        if($cert.friendlyname -match $key) { $global:mycert = $cert }
        if($cert.subject -match $key) { $global:mycert = $cert }
    }
}

new-variable -name userStore -value "currentUser" -option readonly
$global:mycert = $null

if($help) { "Printing help now..." ; funHelp }
if($listcerts) { "Listing certificates in $store" ; funcert }
if(!$cert) {
    "A certificate is required..." ;
    funhelp
}

Findcert($cert)

$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store, $userStore
$objstore.Open("ReadWrite")
$objstore.remove($mycert)
$objstore.Close()

```

Summary

In this chapter, we examined the various ways network administrators commonly work with certificate services. We began by searching the certificate store to locate a specific certificate. Next, we examined using the .NET Framework classes to list all of the certificates in a specific namespace and then looked at locating expired and soon-to-expire certificates.

We then examined the tasks involved in managing certificates, first looking at inspecting a certificate file, and then moving on to importing certificates. We concluded the chapter by examining the process of deleting certificates from the certificate store.