Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/secnetlpMSDN.asp

Roadmap

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
- Microsoft® .NET Framework version 1.0
- ASP.NET
- Enterprise Services
- Web services
- .NET Remoting
- ADO.NET
- Visual Studio® .NET
- SQL™ Server
- Windows® 2000

Summary: This guide presents a practical, scenario driven approach to designing and building secure ASP.NET applications for Windows 2000 and version 1.0 of the .NET Framework. It focuses on the key elements of authentication, authorization, and secure communication within and across the tiers of distributed .NET Web applications. (This roadmap: 6 printed pages; the entire guide: 608 printed pages)

Download

Download Building Secure ASP.NET Applications in .pdf format. (1.67 MB, 608 printed pages)

Contents

Recommendations and sample code in the guide were built and tested using Visual Studio .NET Version 1.0 and validated on servers running Windows 2000 Advanced Server SP 3, .NET Framework SP 2, and SQL Server 2000 SP 2.

What This Guide Is About

This guide focuses on:

- Authentication (to identify the clients of your application)
- Authorization (to provide access controls for those clients)

- Secure communication (to ensure that messages remain private and are not altered by unauthorized parties)

Why authentication, authorization, and secure communication?

Security is a broad topic. Research has shown that early design of authentication and authorization eliminates a high percentage of application vulnerabilities. Secure communication is an integral part of securing your distributed application to protect sensitive data, including credentials, passed to and from your application, and between application tiers.

There are many technologies used to build .NET Web applications. To build effective application-level authentication and authorization strategies, you need to understand how to fine-tune the various security features within each product and technology area, and how to make them work together to provide an effective, defense-in-depth security strategy. This guide will help you do just that.

Figure 1 summarizes the various technologies discussed throughout the guide.

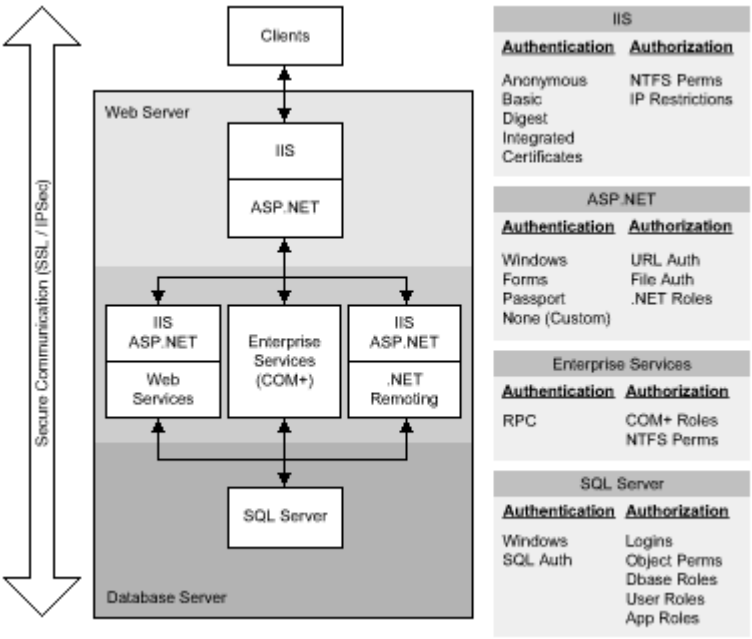

Figure 1. .NET Web application security

The guide is divided into four parts. The aim is to provide a logical partitioning, which will help you to more easily digest the content.

Part I, Security Models

Part I of the guide provides a foundation for the rest of the guide. Familiarity with the concepts, principles, and technologies introduced in Part I will allow you to extract maximum value from the remainder of the guide. Part I contains the following chapters.

- Chapter 1: Introduction

  This chapter highlights the goals of the guide, introduces key terminology, and presents a set of core principles that apply to the guidance presented in later chapters.

- Chapter 2: Security Model for ASP.NET Applications

  This chapter describes the common characteristics of .NET Web applications from a security perspective and introduces the .NET Web application security model. It also introduces the set of core implementation technologies that you will use to build secure .NET Web applications.

- Chapter 3: Authentication and Authorization

  Designing a coherent authentication and authorization strategy across your application's multiple tiers is a critical task. This chapter provides guidance to help you develop an appropriate strategy for your particular application scenario. It will help you choose the most appropriate authentication and authorization technique and apply them at the correct places in your application.

- Chapter 4: Secure Communication

  This chapter introduces the two core technologies that can be used to provide message confidentiality and message integrity for data that flows across the network between clients and servers on the Internet and corporate intranet. These are SSL and IPSec. This chapter also discusses RPC encryption, which can be used to secure the communication with remote serviced components.

Part II, Application Scenarios

Most applications can be categorized as intranet, extranet, or Internet applications. This part of the guide presents a set of common application scenarios, each of which falls into one of those categories. The key characteristics of each scenario are described and the potential security threats analyzed.

You are then shown how to configure and implement the most appropriate authentication, authorization, and secure communication strategy for each application scenario.

- Chapter 5: Intranet Security

  This chapter describes how to secure common intranet application scenarios.

- Chapter 6: Extranet Security

  This chapter describes how to secure common extranet application scenarios.

- Chapter 7: Internet Security

  This chapter describes how to secure common Internet application scenarios.

Part III, Securing the Tiers

This part of the guide contains detailed drill-down information that relates to the individual tiers and technologies associated with secure .NET Web applications. Part III contains the following chapters.

- Chapter 8: ASP.NET Security

  This chapter provides in-depth security recommendations for ASP.NET Web applications. It describes how to implement Forms and Windows authentication and how to perform authorization using the various gatekeepers supported by ASP.NET. Among many other topics, it also discusses how to store secrets, how to use the correct process identity, and how to access network resources such as remote databases by using Windows authentication.

- Chapter 9: Enterprise Services Security

  This chapter explains how to secure business functionality in serviced components contained within Enterprise Services applications. It shows you how and when to use Enterprise Services (COM+) roles for authorization, and how to configure RPC authentication and impersonation. It also shows you how to securely call serviced components from an ASP.NET Web application and how to identify and flow the original caller's security context through a middle tier serviced component.

- Chapter 10: Web Services Security

  This chapter focuses on platform-level security for Web services using the underlying features of Internet Information Services (IIS) and ASP.NET. For message-level security, Microsoft is developing the Web Services Development Kit, which allows you to build security solutions that conform to the WS-Security specification, part of the Global XML Architecture (GXA) initiative.

- Chapter 11: Remoting Security

  The .NET Framework provides a remoting infrastructure that allows clients to communicate with objects, hosted in remote application domains and processes or on remote computers. This chapter shows you how to implement secure .NET Remoting solutions.

- Chapter 12: Data Access Security

  This chapter presents recommendations and guidance that will help you develop a secure data access strategy. Topics covered include using Windows authentication from ASP.NET to the database, securing connection strings, storing credentials securely in a database, protecting against SQL injection attacks, and using database roles.

Part IV, Reference

This reference part of the guide contains supplementary information to help further your understanding of the techniques, strategies, and security solutions presented in earlier chapters.

- Chapter 13: Troubleshooting Security

  This chapter presents a set of troubleshooting tips, techniques, and tools to help diagnose security related issues.

- How Tos

  This section contains a series of step-by-step How-to articles that walk you through many of the solution techniques discussed in earlier chapters.

- Base Configuration

  This section lists the hardware and software used during the development and testing of the guide.

- Configuration Stores and Tools

  This section summarizes the configuration stores used by the various authentication, authorization, and secure communication services and lists the associated maintenance tools.

- Reference Hub

  This section provides a set of links to useful articles and Web sites that provide additional background information about the core topics discussed throughout the guide.

- How Does It Work?

  This section provides supplementary information that details how particular technologies work.

- ASP.NET Identity Matrix

  This section summarizes (with examples) the variables available to ASP.NET Web applications, Web services, and remote components hosted within ASP.NET that provide caller, thread, and process-level identity information.

- Cryptography and Certificates

  This section includes supplementary background information about cryptography and certificates.

- .NET Web Application Security

  This section provides a diagram that shows the authentication, authorization, and secure communication services available across the tiers of an ASP.NET application.

- Glossary

A glossary of security terminology used throughout the guide.

Who Should Read This Guide?

If you are a middleware developer or architect, who plans to build, or is currently building .NET Web applications using one or more of the following technologies, you should read this guide.

- ASP.NET
- Web services
- Enterprise Services
- Remoting
- ADO.NET

What You Must Know

To most effectively use this guide to design and build secure .NET Web applications, you should already have some familiarity and experience with .NET development techniques and technologies. You should be familiar with distributed application architecture and if you have already implemented .NET Web application solutions, you should know your own application architecture and deployment pattern.

Feedback and Support

Questions? Comments? Suggestions? For feedback on this security guide, please send e-mail to secguide@microsoft.com.

The security guide is designed to help you build secure .NET distributed applications. The sample code and guidance is provided as-is. Support is available through Microsoft Product Support for a fee.

Collaborators

Many thanks to the following contributors and reviewers:

Manish Prabhu, Jesus Ruiz-Scougall, Jonathan Hawkins and Doug Purdy, Keith Ballinger, Yann Christensen and Alexei Vopilov, Laura Barsan, Greg Fee, Greg Singleton, Sebastian Lange, Tarik Soulami, Erik Olson, Caesar Samsi, Riyaz Pishori, Shannon Pahl, Ron Jacobs, Dave McPherson, Christopher Brown, John Banes, Joel Scambray, Girish Chander, William Zentmayer, Shantanu Sarkar, Carl Nolan, Samuel Melendez, Jacquelyn Schmidt, Steve Busby, Len Cardinal, Monica DeZulueta, Paula Paul, Ed Draper, Sean Finnegan, David Alberto, Kenny Jones, Doug Orange, Alexey Yeltsov, Martin Kohlleppel, Joel Yoker, Jay Nanduri, Ilia Fortunov, Aaron Margosis (MCS), Venkat Chilakala, John Allen, Jeremy Bostron, Martin Petersen-Frey, Karl Westerholm, Jayaprakasam Siddian Thirunavukkarasu, Wade Mascia, Ryan Kivett, Sarath Mallavarapu, Jerry Bryant, Peter Kyte, Philip Teale, Ram Sunkara, Shaun Hayes, Eric Schmidt, Michael Howard, Rich Benack, Carlos Lyons, Ted Kehl, Peter Dampier, Mike Sherrill, Devendra Tiwari, Tavi Siochi, Per Vonge Nielsen, Andrew Mason, Edward Jezierski, Sandy Khaund, Edward Lafferty, Peter M. Clift, John Munyon, Chris Sfanos, Mohammad Al-Sabt, Anandha Murukan (Satyam), Keith Brown (DevelopMentor), Andy Eunson, John Langley (KANA Software), Kurt Dillard, Christof Sprenger, J.K.Meadows, David Alberto, Bernard Chen (Sapient)

At a Glance

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

Summary: This section allows you to quickly see the scope and coverage of the individual chapters in the guide. (12 printed pages)

Contents

Chapter 1: Introduction

This chapter highlights the goals of the guide, introduces key terminology and presents a set of core principles that apply to the guidance presented in later chapters.

Chapter 2: Security Model for ASP.NET Applications

This chapter describes the common characteristics of.NET Web applications from a security perspective and introduces the .NET Web application security model. It also introduces the set of core implementation technologies that you will use to build secure .NET Web applications.

The full range of gatekeepers that allow you to develop defense-in-depth security strategies are also introduced and the concept of principal-based authorization, using principal and identity objects is explained.

This chapter will help you answer the following questions:

- What are the typical deployment patterns adopted by .NET Web applications?
- What security features are provided by the various technologies that I use to build .NET Web applications?
- What gatekeepers should I be aware of and how do I use them to provide a defense-in-depth security strategy?
- What are principal and identity objects and why are they so significant?
- How does .NET security relate to Windows security?

Chapter 3: Authentication and Authorization

Designing a coherent authentication and authorization strategy across your application's multiple tiers is a critical task. This chapter provides guidance to help you develop an appropriate strategy for your particular application scenario. It will help you choose the most appropriate authentication and authorization technique and apply them at the correct places in your application.

Read this chapter to learn how to:

- Choose an appropriate authentication mechanism to identify users.
- Develop an effective authorization strategy.
- Choose an appropriate type of role-based security.
- Compare and contrast .NET roles with Enterprise Services (COM+) roles.
- Use database roles.
- Choose between the trusted subsystem resource access model and the impersonation/delegation model, which is used to flow the original caller's security context at the operating system level throughout an application's multiple tiers.

These two core resource access models are shown below in Figure 1 and Figure 2.

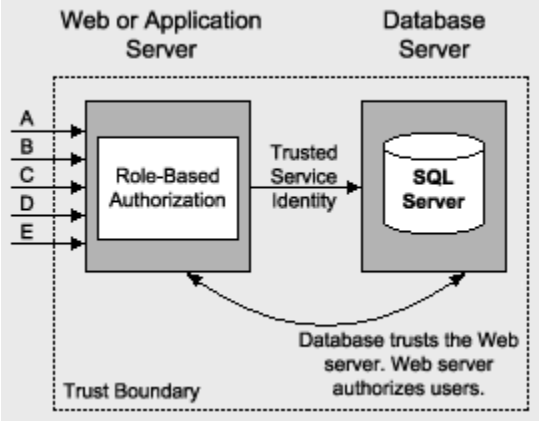

Figure 1. The Trusted Subsystem model

With the trusted subsystem model:

- Downstream resource access is performed using a fixed *trusted* identity and security context.
- The downstream resource manager (for example, database) *trusts* the upstream application to properly authenticate and authorize callers.
- The resource manager authorizes the application to access resources. Original callers are not authorized to directly access the resource manager.
- A trust boundary exists between the downstream and upstream components.
- Original caller identity (for auditing) flows at the application (not operating system) level.

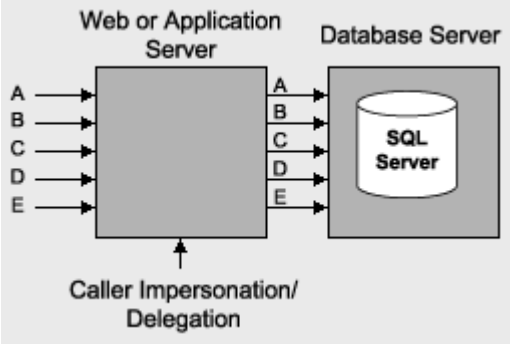

Figure 2. The impersonation / delegation model

With the impersonation/delegation model:

- Downstream resource access is performed using the original caller's security context.
- The downstream resource manager (for example, database) authorizes individual callers.
- The original caller identity flows at the operating system and is available for platform level auditing and per caller authorization.

Chapter 4: Secure Communication

This chapter introduces the two core technologies that can be used to provide message confidentiality and message integrity for data that flows across the network between clients and servers on the Internet and corporate intranet. These are SSL and IPSec. It also discusses RPC encryption that can be used to secure the communication with remote serviced components.

Read this chapter to learn how to:

- Apply secure communication techniques throughout the various tiers of your application.
- Choose between SSL and IPSec.
- Configure secure communication.
- Use RPC encryption.

The chapter addresses the need to provide secure communication channels between your application's various physical tiers as shown in Figure 3.

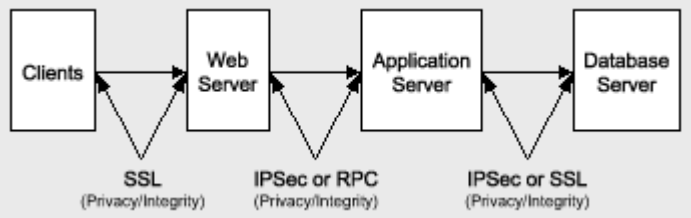

Figure 3. A typical Web deployment model, with secure communications

Chapter 5: Intranet Security

This chapter presents a set of common intranet application scenarios and for each one presents recommended security configurations. In each case, the configuration steps necessary to build the secure solution are presented, together with analysis and related scenario variations.

The application scenarios covered in this chapter are:

- ASP.NET to SQL Server

  This scenario is shown in Figure 4.

- ASP.NET to Enterprise Services to SQL Server
- ASP.NET to Web services to SQL Server
- ASP.NET to Remoting to SQL Server
- Flowing the original caller to the database

  This includes multi-tier Kerberos delegation scenarios, as shown in Figure 5.

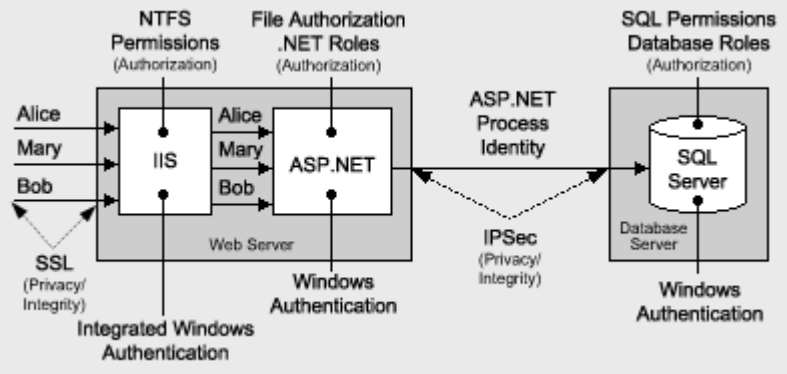

Figure 4. Security configuration for ASP.NET to remote SQL Server scenarios

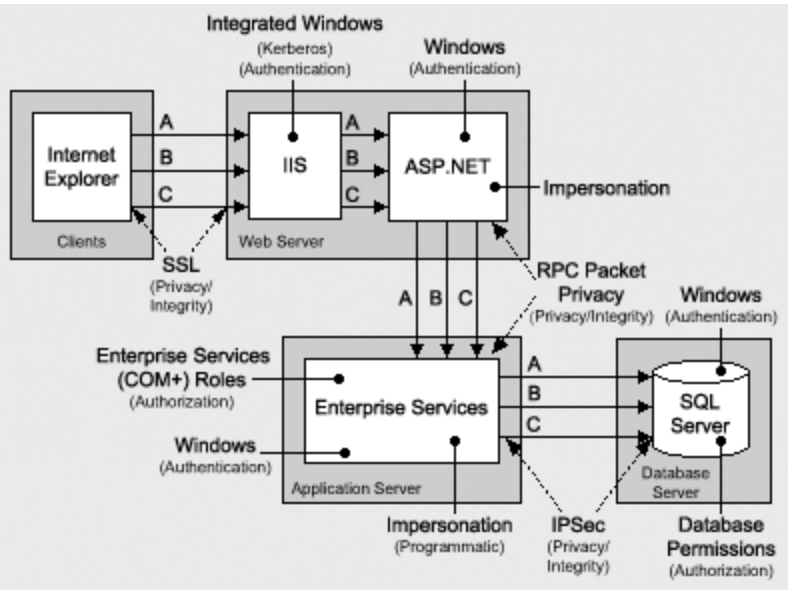

Figure 5. Security configuration for ASP.NET to remote Enterprise Services to remote SQL Server Kerberos delegation scenario

Read this chapter to lean how to:

- Use the local ASPNET account to make calls from an ASP.NET Web application to a remote SQL Server database.
- Establish trusted database connections to SQL Server using Windows authentication.
- Authorize database access with SQL Server user-defined database roles.
- Avoid storing credentials within your application.
- Secure sensitive data with a combination of SSL and IPSec.
- Implement Kerberos delegation to flow the original caller's security context across multiple application tiers to a back-end database.
- Flow the original caller's security context by using Basic authentication.
- Authorize users with a combination of ASP.NET file authorization, URL authorization, .NET roles and Enterprise Services (COM+) roles.
- Effectively use impersonation within an ASP.NET Web application.

Chapter 6: Extranet Security

This chapter presents a set of common extranet application scenarios and for each one presents recommended security configurations, configuration steps and analysis.

This chapter covers the following extranet scenarios.

- Exposing a Web Service (B2B partner exchange)

  This scenario is shown in Figure 6.

- Exposing a Web Application (partner application portal)

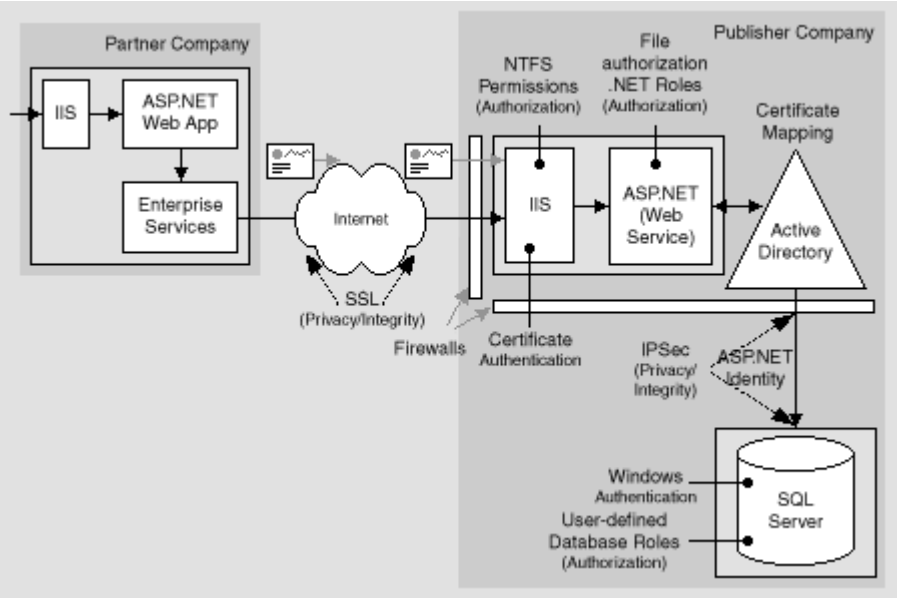

Figure 6. Security configuration for Web Service B2B partner exchange scenario

Read this chapter to lean how to:

- Authenticate partner companies by using client certificate authentication against a dedicated extranet Active Directory.
- Map certificates to Windows accounts.
- Authorize partner companies by using ASP.NET file authorization and .NET roles.
- Use the ASPNET identity to access a remote SQL Server database located on the corporate intranet.

Chapter 7: Internet Security

This chapter presents a set of common Internet application scenarios, and for each one presents recommended security configurations, configuration steps, and analysis.

This chapter covers the following Internet application scenarios:

- ASP.NET to SQL Server
- ASP.NET to Remote Enterprise Services to SQL Server

  This scenario is shown in Figure 7.

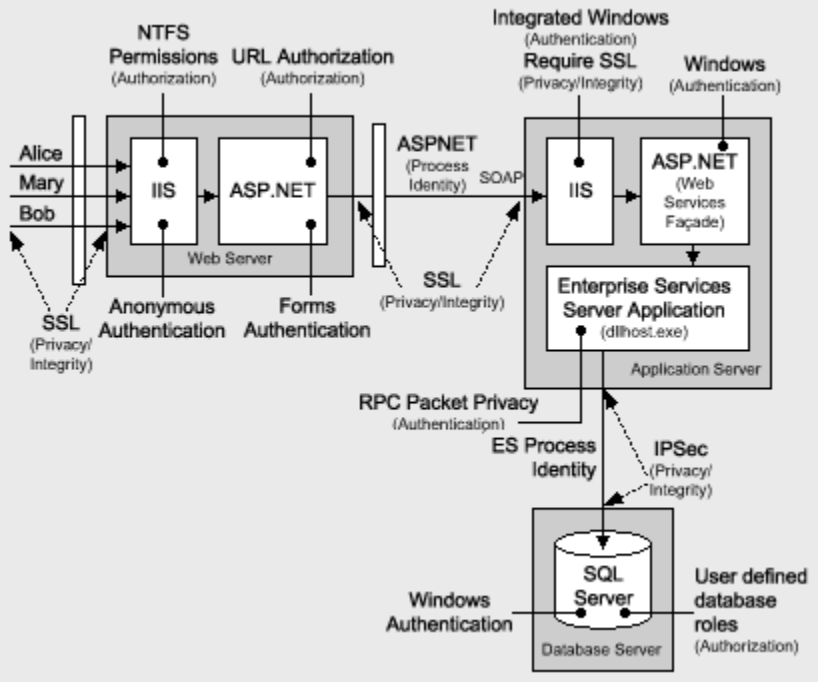

Figure 7. Security configuration for ASP.NET to remote Enterprise Services to SQL Server

Read this chapter to learn how to:

- Use Forms authentication with a SQL Server credential database.
- Avoid storing passwords in the credential database.
- Authorize Internet users with URL Authorization and .NET roles.
- Use Windows authentication from an ASP.NET Web application to SQL Server through a firewall.
- Secure sensitive data with a combination of SSL and IPSec.
- Communicate from an ASP.NET Web application to a remote Enterprise Services application through a firewall by using SOAP.
- Secure calls to serviced component in the application's middle tier.

Chapter 8: ASP.NET Security

This chapter provides in-depth security recommendations for ASP.NET Web applications. This chapter covers the range of authentication, authorization and secure communication services provided by IIS and ASP.NET. These are illustrated in Figure 8.

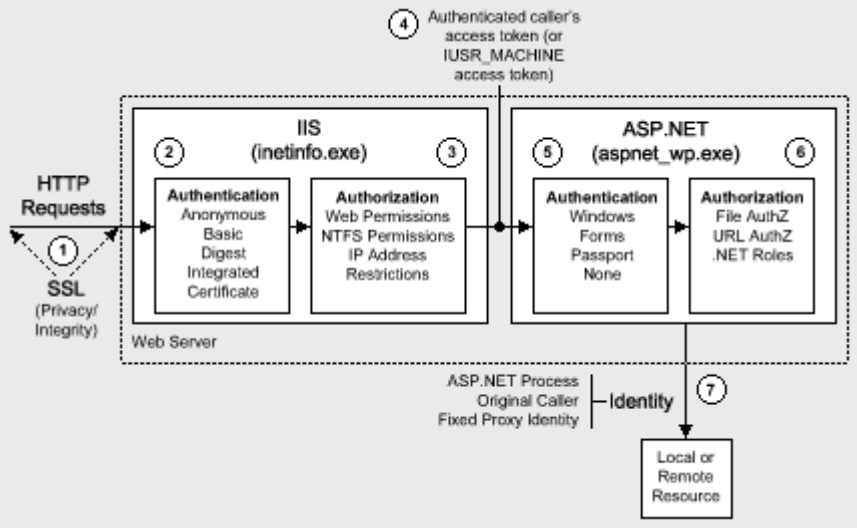

Figure 8. ASP.NET security services

Read this chapter to learn how to:

- Configure the various ASP.NET authentication modes.
- Implement Forms authentication.
- Implement Windows authentication.
- Work with **IPrincipal** and **IIdentity** objects.
- Effectively use the IIS and ASP.NET gatekeepers.
- Configure and use ASP.NET File authorization.
- Configure and use ASP.NET URL authorization.
- Implement declarative, imperative and programmatic role-based security, using principal permission demands and **IPrincipal.IsInRole**.
- Know when and when not to use impersonation within an ASP.NET Web application.
- Choose an appropriate account to run ASP.NET.
- Access local and network resources using the ASP.NET process identity.
- Access remote SQL Server databases using the local ASPNET account.
- Call COM objects from ASP.NET.
- Effectively use the anonymous Internet user account in Web hosting environments.
- Store secrets in an ASP.NET Web application.
- Secure session and view state.
- Configure ASP.NET security in Web Farm scenarios.

Chapter 9: Enterprise Services Security

This chapter explains how to secure business functionality in serviced components contained within Enterprise Services applications. It shows you how and when to use Enterprise Services (COM+) roles for authorization, and how to configure RPC authentication and impersonation. It also shows you how to securely call serviced components from an ASP.NET Web application and how to identify and flow the original caller's security context through a middle tier serviced component.

Figure 9 shows the Enterprise Services security features covered by this chapter.

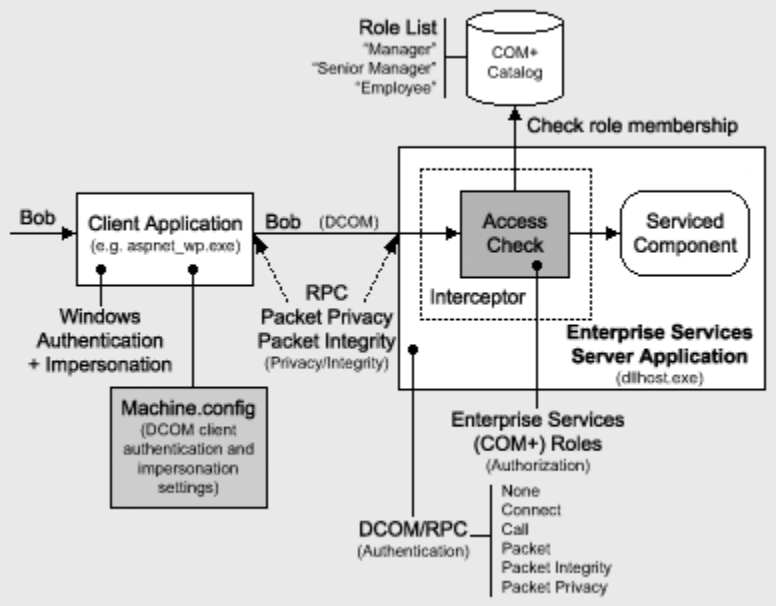

Figure 9. Enterprise Services security overview

Read this chapter to learn how to:

- Configure an Enterprise Services application using .NET attributes.
- Secure server and library applications.
- Choose an appropriate account to run an Enterprise Services server application.
- Implement method level Enterprise Services (COM+) role based security both programmatically and declaratively.
- Configure ASP.NET as a DCOM client.
- Securely call serviced components from ASP.NET.
- Compare Enterprise Services (COM+) roles with .NET roles.
- Identify callers within a serviced component.
- Flow the original caller's security context through an Enterprise Services application by using programmatic impersonation within a serviced component.
- Access local and network resources from a serviced component.
- Use RPC encryption to secure sensitive data passed to and from serviced components.
- Understand the process of RPC authentication level negotiation.
- Use DCOM through firewalls.

Chapter 10: Web Services Security

This chapter focuses on platform level security for Web services using the underlying features of IIS and ASP.NET. For message level security, Microsoft is developing the Web Services Development Kit, which allows you to build security solutions that conform to the WS-Security specification, part of the Global XML Architecture (GXA) initiative.

The ASP.NET Web services platform security architecture is shown in Figure 10.

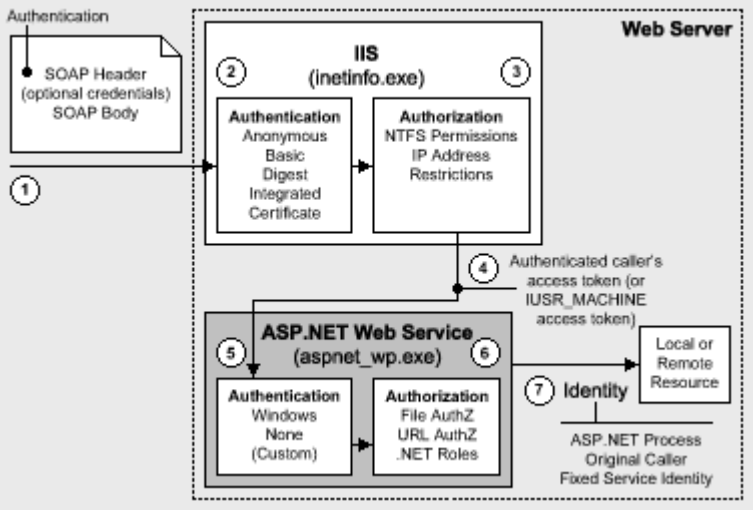

Figure 10. Web services security architecture

Read this chapter to learn how to:

- Implement platform-based Web service security solutions.
- Develop an authentication and authorization strategy for a Web service.
- Use client certificate authentication with Web services.
- Use ASP.NET file authorization, URL authorization, and .NET roles to provide authorization in Web services.
- Flow the original caller's security context through a Web service.
- Call Web services using SSL.
- Access local and network resources from Web services.
- Pass credentials for authentication to a Web service through a Web service proxy.
- Implement the trusted subsystem model for Web services.
- Call COM objects from Web services.

Chapter 11: .NET Remoting Security

The .NET Framework provides a remoting infrastructure that allows clients to communicate with objects, hosted in remote application domains and processes, or on remote computers. This chapter shows you how to implement secure .NET Remoting solutions.

Read this chapter to learn how to:

- Choose an appropriate host for remote components.
- Use all of the available gatekeepers to provide defense-in-depth security.
- Use URL authentication and .NET roles to authorize access to remote components.
- Use File authentication with remoting. This requires you to create a physical .rem or .soap file that corresponds to the remote component's object URI.
- Access local and network resources from a remote component.
- Pass credentials for authentication to a remote component through the remote component proxy object.

- Flow the original caller's security context through a remote component.
- Secure communication to and from remote components using a combination of SSL and IPSec.
- Know when to use remoting and when to use Web services.

Chapter 12: Data Access Security

This chapter presents recommendations and guidance that will help you develop a secure data access strategy. The key issues covered by this chapter are shown in Figure 11. These include storing connection strings securely, using an appropriate identity for database access, securing data passed to and from the database, using an appropriate authentication mechanism and implementing authorization in the database.

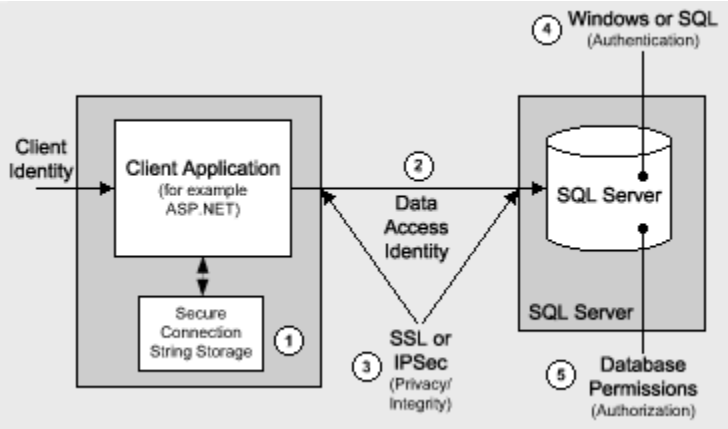

Figure 11. Data Access security overview

Read this chapter to learn how to:

- Use Windows authentication from ASP.NET to your database.
- Secure connection strings.
- Use DPAPI from ASP.NET Web applications to store secrets such as connection strings and credentials.
- Store credentials for authentication securely in a database.
- Validate user input to protect against SQL injection attacks.
- Mitigate the security threats associated with the use of SQL authentication.
- Know which type of database roles to use.
- Compare and contrast database user roles with SQL Server application roles.
- Secure communication to SQL Server using IPSec and also SSL.
- Create a least privilege database account.
- Enable auditing in SQL Server.

Chapter 13: Troubleshooting Security Issues

This chapter provides troubleshooting tips, techniques and tools to help diagnose security related issues. Read this chapter to learn a proven process for effectively troubleshooting security issues you may encounter while building your ASP.NET applications. For example, you'll learn techniques for determining identity in your ASP.NET pages, which can be used to diagnose authentication and access control issues. You'll also learn how to troubleshoot Kerberos authentication. The chapter concludes with a concise list of some of the more useful troubleshooting tools, used by Microsoft support to troubleshoot customer issues.

Reference

Use the supplementary information in this section of the guide to help further your understanding of the techniques, strategies and security solutions presented in earlier chapters. Detailed How To articles provide step-by-step procedures that enable you to implement specific security solutions. It contains the following information:

- Reference Hub
- How To articles
- How Does it Work?
- ASP.NET Identity Matrix
- Base Configuration
- Configuring Security
- Cryptography and Certificates
- .NET Web Application Security Figure
- Glossary

Introduction

J.D. Meier, Alex Mackman, Michael Dunner and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter defines the scope and organization of the guide and highlights its goals. It also introduces key terminology and presents a set of core principles that apply to the guidance presented in later chapters. (7 printed pages)

Contents

Building secure distributed Web applications is challenging. Your application is only as secure as its weakest link. With distributed applications, you have a lot of moving parts and making those parts work together in a secure fashion requires a working knowledge that spans products and technologies.

You already have a lot to consider; integrating various technologies, staying current with technology and keeping a step ahead of the competition. If you don't already know how to build secure applications, can you afford the time and effort to learn? More to the point, can you afford not to?

The Connected Landscape

If you already know how to build secure applications, are you able to apply what you know when you build .NET Web applications? Are you able to apply your knowledge in today's landscape of Web-based distributed applications, where Web services connect businesses to other business and business to customers and where applications offer various degrees of exposure; for example, to users on intranets, extranets and the Internet?

Consider some of the fundamental characteristics of this connected landscape:

- Web services use standards such as SOAP, Extensible Markup Language (XML) and Hypertext Transport Protocol (HTTP), but fundamentally they pass potentially sensitive information using plain text.
- Internet business-to-consumer applications pass sensitive data over the Web.
- Extranet business-to-business applications blur the lines of trust and allow applications to be called by other applications in partner companies.
- Intranet applications are not without their risks considering the sensitive nature of payroll and Human Resource (HR) applications. Such applications are particularly vulnerable to rogue administrators and disgruntled employees.

Scope

This guide focuses on:

- Authentication (to identify the clients of your application)

- Authorization (to provide access controls for those clients)
- Secure communication (to ensure that messages remain private and are not altered by unauthorized parties)

Why authentication, authorization and secure communication?

Security is a broad topic. Research has shown that early design of authentication and authorization eliminates a high percentage of application vulnerabilities. Secure communication is an integral part of securing your distributed application to protect sensitive data, including credentials, passed to and from your application and between application tiers.

What Are the Goals of This Guide?

This guide is not an introduction to security. It is not a security reference for the Microsoft .NET Framework —for that you have the .NET Framework Software Development Kit (SDK) available from MSDN; see the "References" section of this guide for details. This guide picks up where the documentation leaves off and presents a scenario-based approach to sharing recommendations and proven techniques, as gleaned from the field, customer experience and insight from the product teams at Microsoft.

The information in this guide is designed to show you how to:

- Raise the security bar for your application.
- Identify where and how you need to perform authentication.
- Identify where and how you need to perform authorization.
- Identify where and how you need to secure communication both to your application (from your end users) and between application tiers.
- Identify common pitfalls and how to avoid them.
- Identify top risks and their mitigation related to authentication and authorization.
- Avoid opening up security just to make things work.
- Identify not only *how*, but also *when* to use various security features.
- Eliminate FUD (fear, uncertainty and doubt).
- Promote best practices and predictable results.

How You Should Read This Guide

The guide has been developed to be modular. This allows you to pick and choose which chapters to read. For example, if you are interested in learning about the in-depth security features provided by a specific technology, you can jump straight to Part III of the guide (Chapters 8 through 12), which contains in-depth material covering ASP.NET, Enterprise Services, Web Services, .NET Remoting, and data access.

However, you are encouraged to read the early chapters (Chapters 1 through 4) in Part I of the guide first, because these will help you understand the security model and identify the core technologies and security services at your disposal. Application architects should make sure they read Chapter 3, which provides some key insights into designing an authentication and authorization strategy that spans the tiers of your Web application. Part I will provide you with the foundation materials that will allow you to extract maximum benefit from the remainder of the guide.

The intranet, extranet and Internet chapters (Chapters 5 through 7) in Part II of the guide will show you how to secure specific application scenarios. If you know the architecture and deployment pattern that is or will be adopted by your application, use this part of the guide to understand the security issues involved and the basic configuration steps required to secure specific scenarios.

Finally, additional information and reference material in Part IV of the guide will help further your understanding of specific technology areas. It also contains a library of How Tos that enable you to develop working security solutions in the shortest possible time.

Organization of the Guide

The guide is divided into four parts. The aim is to provide a logical partitioning, which will help you to more easily digest the content.

Part I, Security Models

Part 1 of the guide provides a foundation for the rest of the guide. Familiarity with the concepts, principles and technologies introduced in Part 1 will enable you to extract maximum value from the remainder of the guide. Part 1 contains the following chapters.

- Chapter 1, "Introduction"
- Chapter 2, "Security Model for ASP.NET Applications "
- Chapter 3, "Authentication and Authorization"
- Chapter 4, "Secure Communication"

Part II, Application Scenarios

Most applications can be categorized as intranet, extranet or Internet applications. This part of the guide presents a set of common application scenarios, each of which falls into one of the aforementioned categories. The key characteristics of each scenario are described and the potential security threats analyzed.

You are then shown how to configure and implement the most appropriate authentication, authorization and secure communication strategy for each application scenario. Each scenario also contains sections that include a detailed analysis, common pitfalls to watch out for and frequently asked questions (FAQ). Part II contains the following chapters:

- Chapter 5, "Intranet Security"
- Chapter 6, "Extranet Security"
- Chapter 7, "Internet Security"

Part III, Securing the Tiers

This part of the guide contains detailed information that relates to the individual tiers and technologies associated with secure .NET-connected Web applications. Part III contains the following chapters:

- Chapter 8, "ASP.NET Security"
- Chapter 9, "Enterprise Services Security"
- Chapter 10, "Web Services Security"
- Chapter 11, ".NET Remoting Security"
- Chapter 12, "Data Access Security"

Within each chapter, a brief overview of the security architecture as it applies to the particular technology in question is presented. Authentication and authorization strategies are discussed for each technology along with configurable security options, programmatic security options and actionable recommendations of when to use the particular strategy.

Each chapter offers guidance and insight that will allow you to choose and implement the most appropriate authentication, authorization and secure communication option for each technology. In addition, each chapter presents additional information specific to the particular technology. Finally, each chapter concludes with a concise recommendation summary.

Part IV, Reference

This reference part of the guide contains supplementary information to help further your understanding of the techniques, strategies, and security solutions presented in earlier chapters. Detailed How Tos provide step-by-step procedures that enable you to implement specific security solutions. It contains the following information:

- Chapter 13, "Troubleshooting Security"
- How Tos
- "Base Configuration"
- "Configuration Stores and Tools"
- "How Does It Work?"
- "ASP.NET Identity Matrix"
- "Cryptography and Certificates"
- "ASP.NET Security Model"
- "Reference Hub"
- "Glossary"

Key Terminology

This section introduces some key security terminology used throughout the guide. Although a full glossary of terminology is provided within the "Reference" section of this guide, make sure you are very familiar with the following terms:

- **Authentication**. Positively identifying the clients of your application; clients might include end-users, services, processes or computers.
- **Authorization**. Defining what authenticated clients are allowed to see and do within the application.
- **Secure Communications**. Ensuring that messages remain private and unaltered as they cross networks.
- **Impersonation**. This is the technique used by a server application to access resources on behalf of a client. The client's security context is used for access checks performed by the server.
- **Delegation**. An extended form of impersonation that allows a server process that is performing work on behalf of a client, to access resources on a remote computer. This capability is natively provided by Kerberos on Microsoft® Windows® 2000 and later operating systems. Conventional impersonation (for example, that provided by NTLM) allows only a single network hop. When NTLM impersonation is used, the one hop is used between the client and server computers, restricting the server to local resource access while impersonating.
- **Security Context**. Security context is a generic term used to refer to the collection of security settings that affect the security-related behavior of a process or thread. The attributes from a process' logon session and access token combine to form the security context of the process.
- **Identity**. Identity refers to a characteristic of a user or service that can uniquely identify it. For example, this is often a display name, which often takes the form authority/user name.

Principles

There are a number of overarching principles that apply to the guidance presented in later chapters. The following summarizes these principles:

- **Adopt the principle of least privilege**. Processes that run script or execute code should run under a least privileged account to limit the potential damage that can be done if the process is compromised. If a malicious user manages to inject code into a server process, the privileges granted to that process determine to a large degree the types of operations the user is able to perform. Code that requires additional trust (and raised privileges) should be isolated within separate processes.

  The ASP.NET team made a conscious decision to run the ASP.NET account with least privileges (using the ASPNET account). During the beta release of the .NET Framework, ASP.NET ran as SYSTEM, an inherently less secure setting.

- **Use defense in depth**. Place check points within each of the layers and subsystems within your application. The check points are the gatekeepers that ensure that only authenticated and authorized users are able to access the next downstream layer.

- **Don't trust user input**. Applications should thoroughly validate all user input before performing operations with that input. The validation may include filtering out special characters. This preventive measure protects the application against accidental misuse or deliberate attacks by people who are attempting to inject malicious commands into the system. Common examples include SQL injection attacks, script injection and buffer overflow.

- **Use secure defaults**. A common practice among developers is to use reduced security settings, simply to make an application work. If your application demands features that force you to reduce or change default security settings, test the effects and understand the implications before making the change.

- **Don't rely on security by obscurity**. Trying to hide secrets by using misleading variable names or storing them in odd file locations does not provide security. In a game of hide-and-seek, it's better to use platform features or proven techniques for securing your data.

- **Check at the gate**. You don't always need to flow a user's security context to the back end for authorization checks. Often, in a distributed system, this is not the best choice. Checking the client at the gate refers to authorizing the user at the first point of authentication (for example, within the Web application on the Web server), and determining which resources and operations (potentially provided by downstream services) the user should be allowed to access.

  If you design solid authentication and authorization strategies at the gate, you can circumvent the need to delegate the original caller's security context all the way through to your application's data tier.

- **Assume external systems are insecure**. If you don't own it, don't assume security is taken care of for you.

- **Reduce surface area**. Avoid exposing information that is not required. By doing so, you are potentially opening doors that can lead to additional vulnerabilities. Also, handle errors gracefully; don't expose any more information than is required when returning an error message to the end user.

- **Fail to a secure mode**. If your application fails, make sure it does not leave sensitive data unprotected. Also, do not provide too much detail in error messages; meaning don't include details that could help an attacker exploit a vulnerability in your application. Write detailed error information to the Windows event log.

- **Remember you are only as secure as your weakest link**. Security is a concern across all of your application tiers.

- **If you don't use it, disable it**. You can remove potential points of attack by disabling modules and components that your application does not require. For example, if your application doesn't use output caching, then you should disable the ASP.NET output cache module. If a future security vulnerability is found in the module, your application is not threatened.

Summary

This chapter has provided some foundation material to prepare you for the rest of the guide. It has described the goals of the guide and presented its overall structure. Make sure you are familiar with the key terminology and principles introduced in this chapter, because these are used and referenced extensively throughout the forthcoming chapters.

Security Model for ASP.NET Applications

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter describes the common characteristics of .NET Web applications from a security perspective and introduces the .NET Web application security model. It also introduces the set of core implementation technologies that you will use to build secure .NET Web applications. (16 printed pages)

Contents

This chapter introduces .NET Web application security. It provides an overview of the security features and services that span the tiers of a typical .NET Web application.

The goal of the chapter is to:

- Provide a frame of reference for typical .NET Web applications
- Identify the authentication, authorization and secure communication security features provided by the various implementation technologies used to build .NET Web applications
- Identify gatekeepers and gates that can be used in your application to enforce trust boundaries

.NET Web Applications

This section provides a brief introduction to .NET Web applications and describes their characteristics both from a logical and physical viewpoint. It also provides an introduction to the various implementation technologies used to build .NET Web applications.

Logical Tiers

Logical application architecture views any system as a set of cooperating services grouped in the following layers:

- User Services
- Business Services
- Data Services

The value of this logical architecture view is to identify the generic types of services invariably present in any system, to ensure proper segmentation and to drive the definition of interfaces between tiers. This segmentation allows you to make more discreet architecture and design choices when implementing each layer, and to build a more maintainable application.

The layers can be described as follows:

- **User Services** are responsible for the client interaction with the system and provide a common bridge into the core business logic encapsulated by components within the Business Services layer. Traditionally, User Services are associated most often with interactive users. However, they also perform the initial processing of programmatic requests from other systems, where no visible user interface is involved. Authentication and

authorization, the precise nature of which varies depending upon the client type, are typically performed within the User Services layer.

- **Business Services** provide the core functionality of the system and encapsulate business logic. They are independent from the delivery channel and back-end systems or data sources. This provides the stability and flexibility necessary to evolve the system to support new and different channels and back-end systems. Typically, to service a particular business request involves a number of cooperating components within the Business Services layer.

- **Data Services** provide access to data (hosted within the boundaries of the system), and to other (back-end) systems through generic interfaces, which are convenient to use from components within the Business Services layer. Data Services abstract the multitude of back-end systems and data sources, and encapsulate specific access rules and data formats.

The logical classification of service types within a system may correlate with, but is relatively independent from, the possible physical distribution of the components implementing the services.

It is also important to remember that the logical tiers can be identified at any level of aggregation; that is, the tiers can be identified for the system as a whole (in the context of its environment and external interactions) and for any contained subsystem. For example, each remote node that hosts a Web service consists of User Services (handling incoming requests and messages), Business Services and Data Services.

Physical Deployment Models

The three logical service layers described earlier, in no way imply specific numbers of physical tiers. All three logical services may be physically located on the same computer, or they may be spread across multiple computers.

The Web server as an application server

A common deployment pattern for .NET Web applications is to locate business and data access components on the Web server. This minimizes the network hops, which can help performance. This model is shown in Figure 2.1.

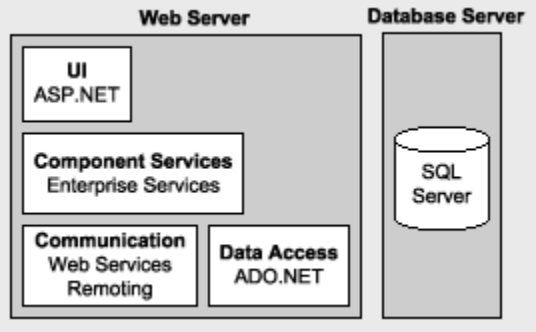

Figure 2.1. The Web server as an application server

Remote application tier

The remote application tier is a common deployment pattern, particularly for Internet scenarios where the Web tier is self-contained within a perimeter network (also known as DMZ, demilitarized zone, and screened subnet) and is separated from end users and the remote application tier with packet filtering firewalls. The remote application tier is shown in Figure 2.2.

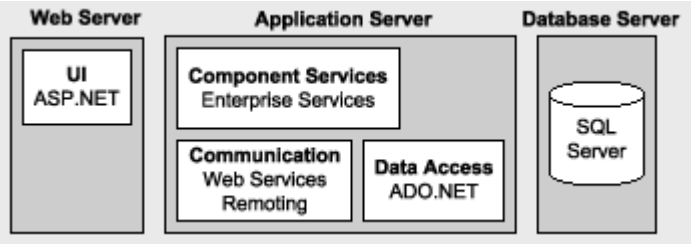

Figure 2.2. The introduction of a remote application tier

Implementation Technologies

.NET Web applications typically implement one or more of the logical services by using the following technologies:

- ASP.NET
- Enterprise Services
- Web services
- .NET Remoting
- ADO.NET and Microsoft® SQL Server™ 2000
- Internet Protocol Security (IPSec)
- Secure Sockets Layer (SSL)

ASP.NET

ASP.NET is typically used to implement User Services. ASP.NET provides a pluggable architecture that can be used to build Web pages. For more information about ASP.NET, see the following resources:

- Chapter 8, ASP.NET Security
- ASP.NET in the "Reference Hub" section of this guide

Enterprise Services

Enterprise Services provide infrastructure-level services to applications. These include distributed transactions and resource management services such as object pooling for .NET components. For more information about Enterprise Services, see the following resources:

- Chapter 9, Enterprise Services Security
- Understanding Enterprise Services (COM+) in .NET on MSDN®
- Enterprise Services in the "Reference Hub" section of this guide

Web Services

Web services enable the exchange of data and the remote invocation of application logic using SOAP-based message exchanges to move data through firewalls and between heterogeneous systems. For more information about Web services, see the following resources:

- Chapter 10, Web Services Security
- XML Web Services Development Center on MSDN
- Web Services in the "Reference Hub" section of this guide

.NET Remoting

.NET Remoting provides a framework for accessing distributed objects across process and machine boundaries. For more information about .NET Remoting, see the following resources:

- Chapter 11, .NET Remoting Security
- Remoting in the "Reference Hub" section of this guide

ADO.NET and SQL Server 2000

ADO.NET provides data access services. It is designed from the ground up for distributed Web applications, and it has rich support for the disconnected scenarios inherently associated with Web applications. For more information about ADO.NET, see the following resources:

- Chapter 12, Data Access Security
- ADO.NET in the "Reference Hub" section of this guide

SQL Server provides integrated security that uses the operating system authentication mechanisms (Kerberos or NTLM). Authorization is provided by logons and granular permissions that can be applied to individual database objects. For more information about SQL Server 2000, see the following resources:

- Chapter 12, Data Access Security

Internet Protocol Security (IPSec)

IPSec provides point-to-point, transport level encryption and authentication services. For more information about IPSec, see the following resources:

- Chapter 4, Secure Communication.
- *IPSec–The New Security Standard for the Internet, Intranets and Virtual Private Networks* by Naganand Doraswamy and Dan Harkins (Prentice Hall PTR, ISBN; ISBN: 0-13-011898-2); Chapter 4 is available on TechNet.

Secure Sockets Layer (SSL)

SSL provides a point-to-point secure communication channel. Data sent over the channel is encrypted. For more information about SSL, see the following resources:

- Chapter 4, Secure Communication
- *Microsoft® Windows® 2000 and IIS 5.0 Administrator's Pocket Consultant* (Microsoft Press, ISBN: 0-7356-1024-X); Chapter 6 is available on TechNet

Security Architecture

Figure 2.3 shows the remote application tier model together with the set of security services provided by the various technologies introduced earlier. Authentication and authorization occurs at many individual points throughout the tiers. These services are provided primarily by Internet Information Services (IIS), ASP.NET, Enterprise Services and SQL Server. Secure communication channels are also applied throughout the tiers and stretch from the client browser or device, right through to the database. Channels are secured with a combination of Secure Sockets Layer (SSL) or IPSec.

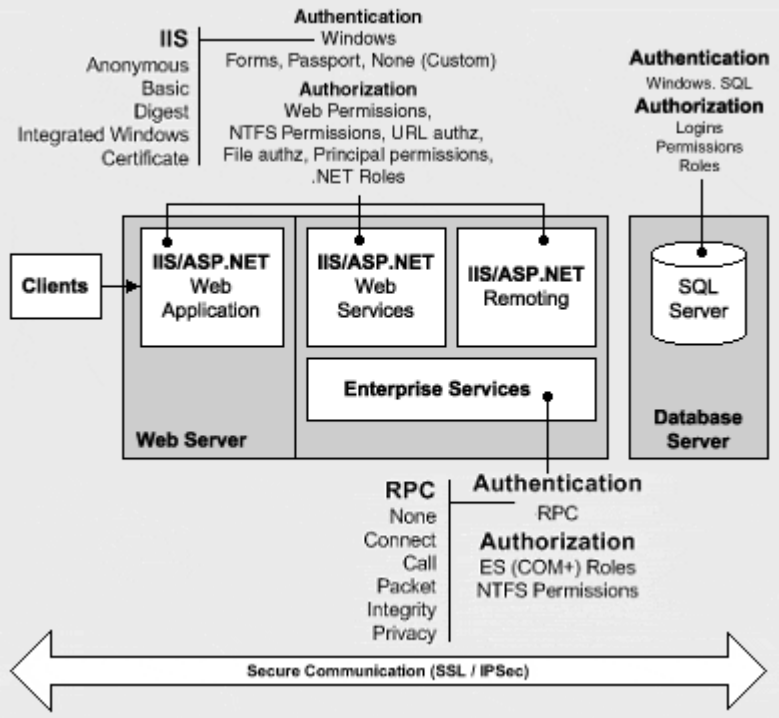

Figure 2.3. Security architecture

Security Across the Tiers

The authentication, authorization, and secure communication features provided by the technologies discussed earlier are summarized in Table 2.1.

Table 2.1. Security features

Technology	Authentication	Authorization	Secure Communication
IIS	Anonymous Basic Digest Windows Integrated (Kerberos/NTLM) Certificate	IP/DNS Address Restrictions Web Permissions NTFS Permissions; Windows Access Control Lists (ACLs) on requested files	SSL
ASP.NET	None (Custom) Windows Forms Passport	File Authorization URL Authorization Principal Permissions .NET Roles	
Web services	Windows None (Custom) Message level authentication	File Authorization URL Authorization Principal Permissions .NET Roles	SSL and Message level encryption
Remoting	Windows	File Authorization URL Authorization Principal Permissions .NET Roles	SSL and message level encryption
Enterprise Services	Windows	Enterprise Services (COM+) Roles NTFS Permissions	Remote Procedure Call (RPC) Encryption

SQL Server 2000	Windows (Kerberos/NTLM) SQL authentication	Server logins Database logins Fixed database roles User defined roles Application roles Object permissions	SSL
Windows 2000	Kerberos NTLM	Windows ACLs	IPSec

Authentication

The .NET Framework on Windows 2000 provides the following authentication options:

- ASP.NET Authentication Modes
- Enterprise Services Authentication
- SQL Server Authentication

ASP.NET authentication modes

ASP.NET authentication modes include Windows, Forms, Passport and None.

- **Windows authentication**. With this authentication mode, ASP.NET relies on IIS to authenticate users and create a Windows access token to represent the authenticated identity. IIS provides the following authentication mechanisms:
  - **Basic authentication**. Basic authentication requires the user to supply credentials in the form of a user name and password to prove their identity. It is a proposed Internet standard based on RFC 2617. Both Netscape Navigator and Microsoft Internet Explorer support Basic authentication. The user's credentials are transmitted from the browser to the Web server in an unencrypted Base64 encoded format. Because the Web server obtains the user's credentials unencrypted, the Web server can issue remote calls (for example, to access remote computers and resources) using the user's credentials.

    Note Basic authentication should only be used in conjunction with a secure channel (typically established by using SSL). Otherwise, user names and passwords can be easily stolen with network monitoring software. If you use Basic authentication you should use SSL on all pages (not just a logon page), because credentials are passed on all subsequent requests. For more information about using Basic authentication with SSL, see Chapter 8, "ASP.NET Security."

  - **Digest authentication**. Digest authentication, introduced with IIS 5.0, is similar to Basic authentication except that instead of transmitting the user's credentials unencrypted from the browser to the Web server, it transmits a hash of the credentials. As a result it is more secure, although it requires an Internet Explorer 5.0 or later client and specific server configuration.
  - **Integrated Windows authentication**. Integrated Windows Authentication (Kerberos or NTLM depending upon the client and server configuration) uses a cryptographic exchange with the user's Internet Explorer Web browser to confirm the identity of the user. It is supported only by Internet Explorer (and not by Netscape Navigator), and as a result tends to be used only in intranet scenarios, where the client software can be controlled. It is used only by the Web server if either anonymous access is disabled or if anonymous access is denied through Windows file system permissions.
  - **Certificate authentication**. Certificate authentication uses client certificates to positively identify users. The client certificate is passed by the user's browser (or client application) to the Web server. (In the case of Web services, the Web services client passes the certificate by means of the ClientCertificates property of the HttpWebRequest object). The Web server then extracts the user's identity from the certificate. This approach relies on a client certificate being installed on the user's computer and as a result tends to be used mostly in intranet or extranet scenarios where the user population is well known and controlled. IIS, upon receipt of a client certificate, can map the certificate to a Windows account.
  - **Anonymous authentication**. If you do not need to authenticate your clients (or you implement a custom authentication scheme), IIS can be configured for Anonymous authentication. In this event, the

Web server creates a Windows access token to represent all anonymous users with the same anonymous (or guest) account. The default anonymous account is IUSR_MACHINENAME, where MACHINENAME is the NetBIOS name of your computer specified at install time.

- **Passport authentication**. With this authentication mode, ASP.NET uses the centralized authentication services of Microsoft Passport. ASP.NET provides a convenient wrapper around functionality exposed by the Microsoft Passport Software Development Kit (SDK), which must be installed on the Web server.

- **Forms authentication**. This approach uses client-side redirection to forward unauthenticated users to a specified HTML form that allows them to enter their credentials (typically user name and password). These credentials are then validated and an authentication ticket is generated and returned to the client. The authentication ticket maintains the user identity and optionally a list of roles that the user is a member of for the duration of the user's session.

  Forms authentication is sometimes used solely for Web site personalization. In this case, you need write little custom code because ASP.NET handles much of the process automatically with simple configuration. For personalization scenarios, the cookie needs to hold only the user name.

  > **Note** Forms authentication sends the user name and password to the Web server in plain text. As a result, you should use Forms authentication in conjunction with a channel secured by SSL. For continued protection of the authentication cookie transmitted on subsequent requests, you should consider using SSL for all pages within your application and not just the logon page.

- **None**. None indicates that you either don't want to authenticate users or that you are using a custom authentication protocol.

More information

For more details about ASP.NET authentication, see Chapter 8, ASP.NET Security.

Enterprise Services authentication

Enterprise Services authentication is performed by using the underlying Remote Procedure Call (RPC) transport infrastructure, which in turn uses the operating system Security Service Provider Interface (SSPI). Clients of Enterprise Services applications may be authenticated using Kerberos or NTLM authentication.

A serviced component can be hosted in a Library application or Server application. Library applications are hosted within client processes and as a result assume the client's identity. Server applications run in separate server processes under their own identity. For more information about identity, see the "Identities and Principals" section later in this chapter.

The incoming calls to a serviced component can be authenticated at the following levels:

- **Default**: The default authentication level for the security package is used.
- **None**: No authentication occurs.
- **Connect**: Authentication occurs only when the connection is made.
- **Call**: Authenticates at the start of each remote procedure call.
- **Packet**: Authenticates and verifies that all call data is received.
- **Packet Integrity**: Authenticates and verifies that none of the data has been modified in transit.
- **Packet Privacy**: Authenticates and encrypts the packet, including the data and the sender's identity and signature.

More information

For more information about Enterprise Services authentication, see Chapter 9, Enterprise Services Security.

SQL Server authentication

SQL Server can authenticate users by using Windows authentication (NTLM or Kerberos) or can use its own built-in authentication scheme referred to as SQL authentication. The following two options are available:

- **SQL Server and Windows**. Clients can connect to an instance of Microsoft SQL Server by using either SQL Server authentication or Windows authentication. This is sometimes referred to as mixed mode authentication.
- **Windows Only**. The user must connect to the instance of Microsoft SQL Server by using Windows authentication.

More information

The relative merits of each approach are discussed in Chapter 12, "Data Access Security."

Authorization

The .NET Framework on Windows 2000 provides of the following authorization options:

- ASP.NET Authorization Options
- Enterprise Services Authorization
- SQL Server Authorization

ASP.NET authorization options

ASP.NET authorization options can be used by ASP.NET Web applications, Web services and remote components. ASP.NET provides the following authorization options:

- **URL Authorization**. This is an authorization mechanism, configured by settings within machine and application configuration files. URL Authorization allows you to restrict access to specific files and folders within your application's Uniform Resource Identifier (URI) namespace. For example, you can selectively deny or allow access to specific files or folders (addressed by means of a URL) to nominated users. You can also restrict access based on the user's role membership and the type of HTTP verb used to issue a request (GET, POST, and so on).

  URL Authorization requires an authenticated identity. This can be obtained by a Windows or ticket-based authentication scheme.

- **File Authorization**. File authorization applies only if you use one of the IIS-supplied Windows authentication mechanisms to authenticate callers and ASP.NET is configured for Windows authentication.

  You can use it to restrict access to specified files on a Web server. Access permissions are determined by Windows ACLs attached to the files.

- **Principal Permission Demands**. Principal permission demands can be used (declaratively or programmatically) as an additional fine-grained access control mechanism. They allow you to control access to classes, methods or individual code blocks based on the identity and group membership of individual users.
- **NET Roles**. .NET roles are used to group together users who have the same permissions within your application. They are conceptually similar to previous role-based implementations, for example Windows groups and COM+ roles. However, unlike these earlier approaches, .NET roles do not require authenticated Windows identities and can be used with ticket-based authentication schemes such as Forms authentication.

  .NET roles can be used to control access to resources and operations and they can be configured both declaratively and programmatically.

More information

For more information about ASP.NET authorization, see Chapter 8, ASP.NET Security.

Enterprise Services authorization

Access to functionality contained in serviced components within Enterprise Services applications is governed by Enterprise Services role membership. These are different from .NET roles and can contain Windows group or user accounts. Role membership is defined within the COM+ catalog and is administered by using the Component Services tool.

More information

For more information about Enterprise Services authorization, see Chapter 9, Enterprise Services Security.

SQL Server authorization

SQL Server allows fine-grained permissions that can be applied to individual database objects. Permissions may be based on role membership (SQL Server provides fixed database roles, user defined roles and application roles), or permission may be granted to individual Windows user or group accounts.

More information

For more information about SQL Server authorization, see Chapter 12, Data Access Security.

Gatekeepers and Gates

Throughout the remainder of this document, the term *gatekeeper* is used to identify the technology that is responsible for a *gate*. A gate represents an access control point (guarding a resource) within an application. For example, a resource might be an operation (represented by a method on an object) or a database or file system resource.

Each of the core technologies listed earlier provide gatekeepers for access authorization. Requests must pass through a series of gates before being allowed to access the requested resource or operation. The following describes the gates the requests must pass through:

- IIS provides a gate when you authenticate users (that is, you disable Anonymous authentication). IIS Web permissions can be used as an access control mechanism to restrict the capabilities of Web users to access specific files and folders. Unlike NTFS file permissions, Web permissions apply to all Web users, as opposed to individual users or groups. NTFS file permissions provide further restrictions on Web resources such as Web pages, images files, and so on. These restrictions apply to individual users or groups.

  IIS checks Web permissions, followed by NTFS file permissions. A user must be authorized by both mechanisms for them to be able to access the file or folder. A failed Web permission check results in IIS returning an HTTP 403–Access Forbidden response, whereas a failed NTFS permission check results in IIS returning an HTTP 401–Access Denied.

- ASP.NET provides various configurable and programmatic gates. These include URL Authorization, File Authorization, Principal Permission demands, and .NET Roles.
- The Enterprise Services gatekeeper uses Enterprise Services roles to authorize access to business functionality.
- SQL Server 2000 includes a series of gates that include server logins, database logins, and database object permissions.
- Windows 2000 provides gates using ACLs attached to secure resources.

The bottom line is that gatekeepers perform authorization based on the identity of the user or service calling into the gate and attempting to access a specific resource. The value of multiple gates is in-depth security with multiple lines of defense. Table 2.2 summaries the set of gatekeepers and identifies for each one the gates that they are responsible for.

Table 2.2. Gatekeepers responsibilities and the gates they provide

Gatekeeper	Gates
Windows Operating System	Logon rights (positive and negative, for example "Deny logon locally") Other privileges (for example "Act as part of the operating system") Access checks against secured resources such as the registry and file system.

	Access checks use ACLs attached to the secure resources, which specify who is and who is not allowed to access the resource and also the types of operation that may be permitted. TCP/IP filtering IP Security
IIS	Authentication (Anonymous, Basic, Digest, Integrated, Certificate) IP address and domain name restrictions (these can be used as an additional line of defense, but should not be relied upon due to the relative ease of spoofing IP addresses). Web permissions NTFS permissions
ASP.NET	URL Authorization File Authorization Principal Permission Demands .NET Roles
Enterprise Services	Windows (NTLM / Kerberos) authentication Enterprise Services (COM+) roles Impersonation levels
Web services	Uses gates provided by IIS and ASP.NET
Remoting	Uses gates provided by the host. If hosted in ASP.NET it uses the gates provided by IIS and ASP.NET. If hosted in a Windows service, then you must develop a custom solution.
ADO.NET	Connection strings. Credentials may be explicit or you may use Windows authentication (for example, if you connect to SQL Server)
SQL Server	Server logins Database logins Database object permissions

By using the various gates throughout the tiers of your application, you can filter out users that should be allowed access to your back-end resources. The scope of access is narrowed by successive gates that become more and more granular as the request proceeds through the application to the back-end resources.

Consider the Internet-based application example using IIS that is shown in Figure 2.4.

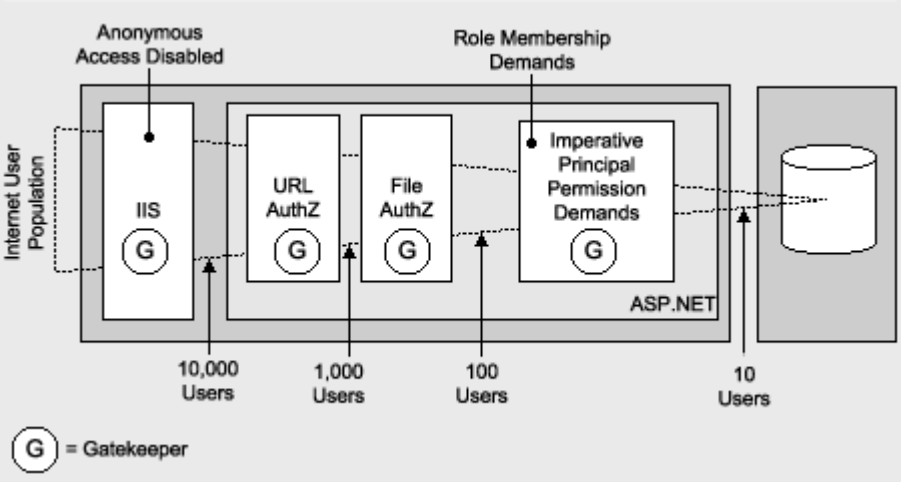

Figure 2.4. Filtering users with gatekeepers

Figure 2.4 illustrates the following:

- You can disable Anonymous authentication in IIS. As a result, only accounts that IIS is able to authenticate are allowed access. This might reduce the potential number of users to 10,000.
- Next, in ASP.NET you use URL Authorization, which might reduce the user count to 1,000 users.
- File authorization might further narrow access down to 100 users.
- Finally, your Web application code might allow only 10 users to access your restricted resource, based on specific role membership.

Identities and Principals

.NET security is layered on top of Windows security. The user centric concept of Windows security is based on security context provided by a logon session while .NET security is based on **IPrincipal** and **IIdentity** objects.

In Windows programming when you want to know the security context code is running under, the identity of the process owner or currently executing thread is consulted. With .NET programming, if you want to query the security context of the current user, you retrieve the current **IPrincipal** object from **Thread.CurrentPrincipal**.

The .NET Framework uses identity and principal objects to represent users when .NET code is running and together they provide the backbone of .NET role-based authorization.

Identity and principal objects must implement the **IIdentity** and **IPrincipal** interfaces respectively. These interfaces are defined within the **System.Security.Principal** namespace. Common interfaces allow the .NET Framework to treat identity and principal objects in a polymorphic fashion, regardless of the underlying implementation details.

The **IPrincipal** interface allows you to test role membership through an **IsInRole** method and also provides access to an associated **IIdentity** object.

```
public interface IPrincipal

{

  bool IsInRole( string role );

  IIdentity Identity {get;}

}
```

The **IIdentity** interface provides additional authentication details such as the name and authentication type.

```
public interface IIdentity

{

  string authenticationType {get;}

  bool IsAuthenticated {get;}

  string Name {get;}

}
```

The .NET Framework supplies a number of concrete implementations of **IPrincipal** and **IIdentity** as shown in Figure 2.5 and described in the following sections.

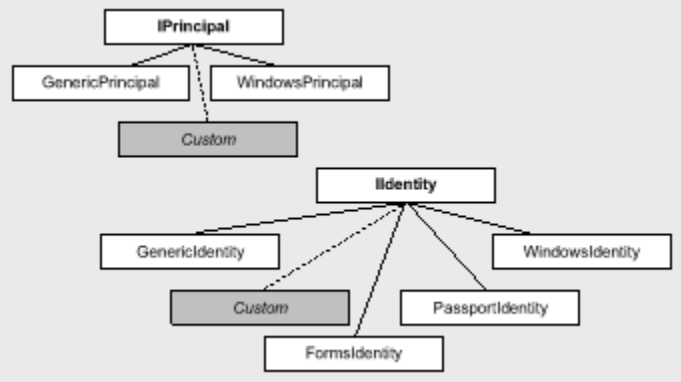

Figure 2.5. IPrincipal and IIdentity implementation classes

WindowsPrincipal and WindowsIdentity

The .NET version of a Windows security context is divided between two classes:

- **WindowsPrincipal**. This class stores the roles associated with the current Windows user. The **WindowsPrincipal** implementation treats Windows groups as roles. The **IPrncipal.IsInRole** method returns true or false based on the user's Windows group membership.
- **WindowsIdentity**. This class holds the identity part of the current user's security context and can be obtained from the static **WindowsIdentity.GetCurrent()** method. This returns a **WindowsIdentity** object that has a **Token** property that returns an **IntPtr** that represents a Windows handle to the access token associated with the current execution thread. This token can then be passed to native Win32® application programming interface (API) functions such as **GetTokenInformation**, **SetTokenInformation**, **CheckTokenMembership** and so on, to retrieve security information about the token.

  Note The static **WindowsIdentity.GetCurrent()** method returns the identity of the currently executing thread, which may or may not be impersonating. This is similar to the Win32 **GetUserName** API.

GenericPrincipal and Associated Identity Objects

These implementations are very simple and are used by applications that do not use Windows authentication and where the application does not need complex representations of a principal. They can be created in code very easily and as a result a certain degree of trust must exist when an application deals with a **GenericPrincipal**.

If you are relying upon using the **IsInRole** method on the **GenericPrincipal** in order to make authorization decisions, you must trust the application that sends you the **GenericPrincipal**. This is in contrast to using **WindowsPrincipal** objects, where you must trust the operating system to provide a valid **WindowsPrincipal** object with an authenticated identity and valid group/role names.

The following types of identity object can be associated with the **GenericPrincipal** class:

- **FormsIdentity**. This class represents an identity that has been authenticated with Forms authentication. It contains a **FormsAuthenticationTicket**, which contains information about the user's authentication session.
- **PassportIdentity**. This class represents an identity that has been authenticated with Passport authentication and contains Passport profile information.
- **GenericIdentity**. This class represents a logical user that is not tied to any particular operating system technology and is typically used in association with custom authentication and authorization mechanisms.

ASP.NET and HttpContext.User

Typically, **Thread.CurrentPrincipal** is checked in .NET code before any authorization decisions are made. ASP.NET, however, provides the authenticated user's security context using **HttpContext.User**.

This property accepts and returns an **IPrincipal** interface. The property contains an authenticated user for the current request. ASP.NET retrieves **HttpContext.User** when it makes authorization decisions.

When you use Windows authentication, the Windows authentication module automatically constructs a **WindowsPrincipal** object and stores it in **HttpContext.User**. If you use other authentication mechanisms such as Forms or Passport, you must construct a **GenericPrincipal** object and store it in **HttpContext.User**.

ASP.NET identities

At any given time during the execution of an ASP.NET Web application, there may be multiple identities present during a single request. These identities include:

- **HttpContext.User** returns an **IPrincipal** object that contains security information for the current Web request. This is the authenticated Web client.
- **WindowsIdentity.GetCurrent()**returns the identity of the security context of the currently executing Win32 thread. By default, this identity is ASPNET; the default account used to run ASP.NET Web applications. However, if the Web application has been configured for impersonation, the identity represents the authenticated user (which if IIS Anonymous authentication is in effect, is IUSR_MACHINE).
- **Thread.CurrentPrincipal** returns the principal of the currently executing .NET thread, which rides on top of the Win32 thread.

More information

- For a detailed analysis of ASP.NET identity for a combination of Web application configurations (both with and without impersonation), see ASP.NET Identity Matrix within the "Reference" section of this guide.
- For more information about creating your own **IPrincipal** implementation, see Chapter 8, ASP.NET Security, and How to Implement IPrincipal in the "Reference" section of this guide.

Remoting and Web Services

In the current version of the .NET Framework, Remoting and Web services do not have their own security model. They both inherit the security feature of IIS and ASP.NET.

Although there is no security built into the Remoting architecture, it was designed with security in mind. It is left up to the developer and/or administrator to incorporate certain levels of security in Remoting applications. Whether or not principal objects are passed across Remoting boundaries depends on the location of the client and remote object, for example:

- **Remoting within the same process**. When remoting is used between objects in the same or separate application domain(s), the remoting infrastructure copies a reference to the **IPrincipal** object associated with the caller's context to the receiver's context.
- **Remoting across processes**. In this case, **IPrincipal** objects are not transmitted between processes. The credentials used to construct the original **IPrincipal** must be transmitted to the remote process, which may be located on a separate computer. This allows the remote computer to construct an appropriate **IPrincipal** object based on the supplied credentials.

Summary

This chapter has introduced the full set of authentication and authorization options provided by the various .NET related technologies. By using multiple gatekeepers throughout your .NET Web application, you will be able to implement a defense-in-depth security strategy. To summarize:

- ASP.NET applications can use the existing security features provided by Windows and IIS.
- A combination of SSL and IPSec can be used to provide secure communications across the layers of a .NET Web application; for example, from browser to database.
- Use SSL to protect the clear text credentials passed across the network when you use Basic or Forms authentication.

- .NET represents users who have been identified with Windows authentication using a combination of the **WindowsPrincipal** and **WindowsIdentity** classes.
- The **GenericPrincipal** and **GenericIdentity** or **FormsIdentity** classes are used to represent users who have been identified with non-Windows authentication schemes, such as Forms authentication.
- You can create your own principal and identity implementations by creating classes that implement **IPrincipal** and **IIdentity**.
- Within ASP.NET Web applications, the **IPrincipal** object that represents the authenticated user is associated with the current HTTP Web request using the **HttpContext.User** property.
- Gates are access control points within your application through which authorized users can access resources or services. Gatekeepers are responsible for controlling access to gates.
- Use multiple gatekeepers to provide a defense-in-depth strategy.

The next chapter, Chapter 3, Authentication and Authorization, provides additional information to help you choose the most appropriate authentication and authorization strategy for your particular application scenario.

Authentication and Authorization

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter provides guidance to help you develop an appropriate authorization strategy for your particular application scenario. It will help you choose the most appropriate authentication and authorization technique and apply them at the correct places in your application. (22 printed pages)

Contents

Designing an authentication and authorization strategy for distributed Web applications is a challenging task. The good news is that proper authentication and authorization design during the early phases of your application development helps to mitigate many top security risks.

This chapter will help you design an appropriate authorization strategy for your application and will also help answer the following key questions:

- Where should I perform authorization and what mechanisms should I use?
- What authentication mechanism should I use?
- Should I use Active Directory® directory service for authentication or should I validate credentials against a custom data store?
- What are the implications and design considerations for heterogeneous and homogenous platforms?
- How should I represent users who do not use the Microsoft® Windows® operating system within my application?
- How should I flow user identity throughout the tiers of my application? When should I use operating system level impersonation/delegation?

When you consider authorization, you must also consider authentication. The two processes go hand in hand for two reasons:

- First, any meaningful authorization policy requires authenticated users.
- Second, the way in which you authenticate users (and specifically the way in which the authenticated user identity is represented within your application) determines the available gatekeepers at your disposal.

  Some gatekeepers such as ASP.NET file authorization, Enterprise Services (COM+) roles and Windows ACLs, require an authenticated Windows identity (in the form of a **WindowsIdentity** object that encapsulates a Windows access token, which defines the caller's security context). Other gatekeepers, such as ASP.NET URL authorization and .NET roles, do not. They simply require an authenticated identity; one that is not necessarily represented by a Windows access token.

Designing an Authentication and Authorization Strategy

The following steps identify a process that will help you develop an authentication and authorization strategy for your application:

1. Identify resources
2. Choose an authorization strategy
3. Choose the identities used for resource access
4. Consider identity flow
5. Choose an authentication approach
6. Decide how to flow identity

Identify Resources

Identify resources that your application needs to expose to clients. Typical resources include:

- Web Server resources such as Web pages, Web services, static resources (HTML pages and images).
- Database resources such as per-user data or application-wide data.
- Network resources such as remote file system resources and data from directory stores such as Active Directory.

You must also identify the system resources that your application needs to access. This is in contrast to resources that are exposed to clients. Examples of system resources include the registry, event logs and configuration files.

Choose an Authorization Strategy

The two basic authorization strategies are:

- **Role based**. Access to operations (typically methods) is secured based on the role membership of the caller. Roles are used to partition your application's user base into sets of users that share the same security privileges within the application; for example, Senior Managers, Managers and Employees .Users are mapped to roles and if the user is authorized to perform the requested operation, the application uses fixed identities with which to access resources. These identities are trusted by the respective resource managers (for example, databases, the file system and so on).
- **Resource based**. Individual resources are secured using Windows ACLs. The application impersonates the caller prior to accessing resources, which allows the operating system to perform standard access checks. All resource access is performed using the original caller's security context. This impersonation approach severely impacts application scalability, because it means that connection pooling cannot be used effectively within the application's middle tier.

In the vast majority of .NET Web applications where scalability is essential, a role-based approach to authorization represents the best choice. For certain smaller scale intranet applications that serve per-user content from resources (such as files) that can be secured with Windows ACLs against individual users, a resource-based approach may be appropriate.

The recommended and common pattern for role-based authorization is:

- Authenticate users within your front-end Web application.
- Map users to roles.
- Authorize access to operations (not directly to resources) based on role membership.
- Access the necessary back-end resources (required to support the requested and authorized operations) by using fixed service identities. The back-end resource managers (for example, databases) *trust* the application to authorize callers and are willing to grant permissions to the trusted service identity or identities.

  For example, a database administrator may grant access permissions exclusively to a specific HR application (but not to individual users).

More information

- For more information about the two contrasting authorization approaches, see Authorization Approaches later in this chapter.
- For more information about role-based authorization and the various types of roles that can be used, see Role-Based Authorization later in this chapter.

Choose the Identities Used for Resource Access

Answer the question, "who will access resources?"

Choose the identity or identities that should be used to access resources across the layers of your application. This includes resources accessed from Web-based applications, and optionally Web services, Enterprise Services and .NET Remoting components. In all cases, the identity used for resource access can be:

- **Original caller's identity**. This assumes an impersonation/delegation model in which the original caller identity can be obtained and then flowed through each layer of your system. The delegation factor is a key criteria used to determine your authentication mechanism.
- **Process identity**. This is the default case (without specific impersonation). Local resource access and downstream calls are made using the current process identity. The feasibility of this approach depends on the boundary being crossed, because the process identity must be recognized by the target system.

  This implies that calls are made in one of the following ways:

  - Within the same Windows security domain
  - Across Windows security domains (using trust and domain accounts, or duplicated user names and passwords where no trust relationship exists)
- **Service account**. This approach uses a (fixed) service account. For example:
  - For database access this might be a fixed SQL user name and password presented by the component connecting to the database.
  - When a fixed Windows identity is required, use an Enterprise Services server application.
- **Custom identity**. When you don't have Windows accounts to work with, you can construct your own identities (using **IPrincipal** and **IIdentity** implementations) that can contain details that relate to your own specific security context. For example, these could include role lists, unique identifiers, or any other type of custom information.

  By implementing your custom identity with **IPrincipal** and **IIdentity** types and placing them in the current Web context (using **HttpContext.User**), you immediately benefit from built-in gatekeepers such as .NET roles and **PrincipalPermission** demands.

Consider Identity Flow

To support per-user authorization, auditing, and per-user data retrieval you may need to flow the original caller's identity through various application tiers and across multiple computer boundaries. For example, if a back-end resource manager needs to perform per-caller authorization, the caller's identity must be passed to that resource manager.

Based on resource manager authorization requirements and the auditing requirements of your system, identify which identities need to be passed through your application.

Choose an Authentication Approach

Two key factors that influence the choice of authentication approach are first and foremost the nature of your application's user base (what types of browsers are they using and do they have Windows accounts), and secondly your application's impersonation/delegation and auditing requirements.

More information

For more detailed considerations that help you to choose an authentication mechanism for your application, see Choosing an Authentication Mechanism later in this chapter.

Decide How to Flow Identity

You can flow identity (to provide security context) at the application level or you can flow identity and security context at the operating system level.

To flow identity at the application level, use method and stored procedure parameters. Application identity flow supports:

- Per-user data retrieval using trusted query parameters

```
SELECT x,y FROM SomeTable WHERE username="bob"
```

- Custom auditing within any application tier

Operating system identity flow supports:

- Platform level auditing (for example, Windows auditing and SQL Server auditing)
- Per-user authorization based on Windows identities

To flow identity at the operating system level, you can use the impersonation/delegation model. In some circumstances you can use Kerberos delegation, while in others (where perhaps the environment does not support Kerberos) you may need to use other approaches such as, using Basic authentication. With Basic authentication, the user's credentials are available to the server application and can be used to access downstream network resources.

More information

For more information about flowing identity and how to obtain an impersonation token with network credentials (that is, supports delegation), see Flowing Identity later in this chapter.

Authorization Approaches

There are two basic approaches to authorization:

- **Role based**. Users are partitioned into application-defined, logical roles. Members of a particular role share the same privileges within the application. Access to operations (typically expressed by method calls) is authorized based on the role-membership of the caller.

  Resources are accessed using fixed identities (such as a Web application's or Web service's process identity). The resource managers trust the application to correctly authorize users and they authorize the *trusted* identity.

- **Resource based**. Individual resources are secured using Windows ACLs. The ACL determines which users are allowed to access the resource and also the types of operation that each user is allowed to perform (read, write, delete and so on).

  Resources are accessed using the original caller's identity (using impersonation).

Role Based

With a role-based (or operations-based) approach to security, access to operations (not back-end resources) is authorized based on the role membership of the caller. Roles (analyzed and defined at application design time) are used as logical containers that group together users who share the same security privileges (or capabilities) within the application. Users are mapped to roles within the application and role membership is used to control access to specific operations (methods) exposed by the application.

Where within your application this role mapping occurs is a key design criterion; for example:

- On one extreme, role mapping might be performed within a back-end resource manager such as a database. This requires the original caller's security context to flow through your application's tiers to the back-end database.
- On the other extreme, role mapping might be performed within your front-end Web application. With this approach, downstream resource managers are accessed using fixed identities that each resource manager authorizes and is willing to trust.
- A third option is to perform role mapping somewhere in between the front-end and back-end tiers; for example, within a middle tier Enterprise Services application.

In multi-tiered Web applications, the use of trusted identities to access back-end resource managers provides greater opportunities for application scalability (thanks to connection pooling). Also, the use of trusted identities alleviates the need to flow the original caller's security context at the operating system level, something that can be difficult (if not impossible in certain scenarios) to achieve.

Resource Based

The resource-based approach to authorization relies on Windows ACLs and the underlying access control mechanics of the operating system. The application impersonates the caller and leaves it to the operating system in conjunction with specific resource managers (the file system, databases, and so on) to perform access checks.

This approach tends to work best for applications that provide access to resources that can be individually secured with Windows ACLs, such as files. An example would be an FTP application or a simple data driven Web application. The approach starts to break down where the requested resource consists of data that needs to be obtained and consolidated from a number of different sources; for example, multiple databases, database tables, external applications or Web services.

The resource-based approach also relies on the original caller's security context flowing through the application to the back-end resource managers. This can require complex configuration and significantly reduces the ability of a multi-tiered application to scale to large numbers of users, because it prevents the efficient use of pooling (for example, database connection pooling) within the application's middle tier.

Resource Access Models

The two contrasting approaches to authorization can be seen within the two most commonly used resource-access security models used by .NET Web applications (and distributed multi-tier applications in general). These are:

- The trusted subsystem model
- The impersonation/delegation model

Each model offers advantages and disadvantages both from a security and scalability perspective. The next sections describe these models.

The Trusted Subsystem Model

With this model, the middle tier service uses a fixed identity to access downstream services and resources. The security context of the original caller does not flow through the service at the operating system level, although the application may choose to flow the original caller's identity at the application level. It may need to do so to support back-end auditing requirements, or to support per-user data access and authorization.

The model name stems from the fact that the downstream service (perhaps a database) trusts the upstream service to authorize callers. Figure 3.1 shows this model. Pay particular attention to the trust boundary. In this example, the database *trusts* the middle tier to authorize callers and allow only authorized callers to access the database using the trusted identity.

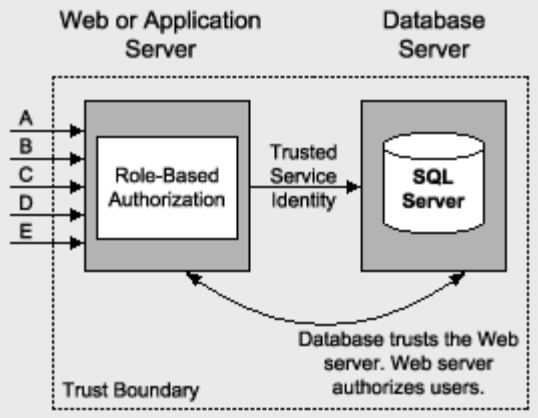

Figure 3.1. The Trusted Subsystem model

The pattern for resource access in the trusted subsystem model is the following:

- Authenticate users
- Map users to roles
- Authorize based on role membership
- Access downstream resource manager using a fixed trusted identity

Fixed identities

The fixed identity used to access downstream systems and resource managers is often provided by a preconfigured Windows account, referred to as a service account. With a Microsoft SQL Server™ resource manager, this implies Windows authentication to SQL Server.

Alternatively, some applications use a nominated SQL account (specified by a user name and password in a connection string) to access SQL Server. In this scenario, the database must be configured for SQL authentication.

For more information about the relative merits of Windows and SQL authentication when communicating with SQL Server, see Chapter 12, Data Access Security.

Using multiple trusted identities

Some resource managers may need to be able to perform slightly more fine-grained authorization, based on the role membership of the caller. For example, you may have two groups of users, one who should be authorized to perform read/write operations and the other read-only operations.

Consider the following approach with SQL Server:

- Create two Windows accounts, one for read operations and one for read/write operations.

  More generally, you have separate accounts to mirror application-specific roles. For example, you might want to use one account for Internet users and another for internal operators and/or administrators.

- Map each account to a SQL Server user-defined database role, and establish the necessary database permissions for each role.
- Map users to roles within your application and use role membership to determine which account to impersonate before connecting to the database.

This approach is shown in Figure 3.2.

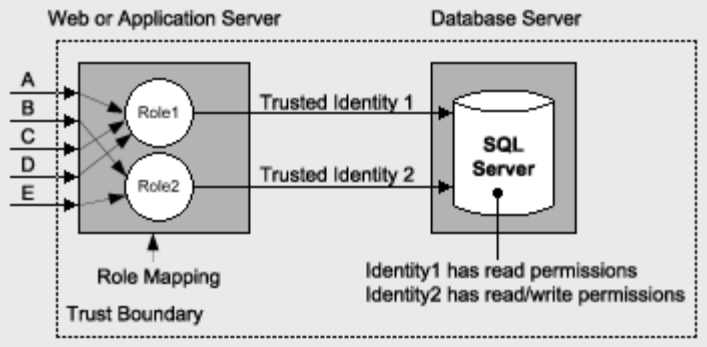

Figure 3.2. Using multiple identities to access a database to support more fine-grained authorization

The Impersonation / Delegation Model

With this model, a service or component (usually somewhere within the logical business services layer) impersonates the client's identity (using operating system-level impersonation) before it accesses the next downstream service. If the next service in line is on the same computer, impersonation is sufficient. Delegation is required if the downstream service is located on a remote computer.

As a result of the delegation, the security context used for the downstream resource access is that of the client. This model is typically used for a couple of reasons:

- It allows the downstream service to perform per-caller authorization using the original caller's identity.
- It allows the downstream service to use operating system-level auditing features.

As a concrete example of this technique, a middle-tier Enterprise Services component might impersonate the caller prior to accessing a database. The database is accessed using a database connection tied to the security context of the original caller. With this model, the database authenticates each and every caller and makes authorization decisions based on permissions assigned to the individual caller's identity (or the Windows group membership of the caller). The impersonation/delegation model is shown in Figure 3.3.

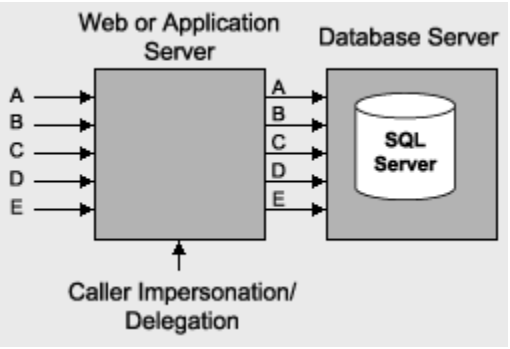

Figure 3.3. The impersonation/delegation model

Choosing a Resource Access Model

The trusted subsystem model is used in the vast majority of Internet applications and large-scale intranet applications, primarily for scalability reasons. The impersonation model tends to be used in smaller-scale applications where scalability is not the primary concern and those applications where auditing (for reasons of non-repudiation) is a critical concern.

Advantage of the impersonation / delegation model

The primary advantage of the impersonation / delegation model is auditing (close to the data). Auditing allows administrators to track which users have attempted to access specific resources. Generally auditing is considered

most authoritative if the audits are generated at the precise time of resource access and by the same routines that access the resource.

The impersonation / delegation model supports this by maintaining the user's security context for downstream resource access. This allows the back-end system to authoritatively log the user and the requested access.

Disadvantages of the impersonation / delegation model

The disadvantages associated with the impersonation / delegation model include:

- **Technology challenges**. Most security service providers don't support delegation, Kerberos is the notable exception.

  Processes that perform impersonation require higher privileges (specifically the Act *as part of the operating system* privilege). (This restriction applies to Windows 2000 and will not apply to Windows Server).

- **Scalability**. The impersonation / delegation model means that you cannot effectively use database connection pooling, because database access is performed by using connections that are tied to the individual security contexts of the original callers. This significantly limits the application's ability to scale to large numbers of users.

- **Increased administration effort**. ACLs on back-end resources need to be maintained in such a way that each user is granted the appropriate level of access. When the number of back-end resources increases (and the number of users increases), a significant administration effort is required to manage ACLs.

Advantages of the trusted subsystem model

The trusted subsystem model offers the following advantages:

- **Scalability**. The trusted subsystem model supports connection pooling, an essential requirement for application scalability. Connection pooling allows multiple clients to reuse available, pooled connections. It works with this model because all back-end resource access uses the security context of the service account, regardless of the caller's identity.

- **Minimizes back-end ACL management**. Only the service account accesses back-end resources (for example, databases). ACLs are configured against this single identity.

- **Users can't access data directly**. In the trusted-subsystem model, only the middle-tier service account is granted access to the back-end resources. As a result, users cannot directly access back-end data without going through the application (and being subjected to application authorization).

Disadvantages of the trusted subsystem model

The trusted-subsystem model suffers from a couple of drawbacks:

- **Auditing**. To perform auditing at the back end, you can explicitly pass (at the application level) the identity of the original caller to the back end, and have the auditing performed there. You have to trust the middle-tier and you do have a potential repudiation risk. Alternatively, you can generate an audit trail in the middle tier and then correlate it with back-end audit trails (for this you must ensure that the server clocks are synchronized).

- **Increased risk from server compromise**. In the trusted-subsystem model, the middle-tier service is granted broad access to back-end resources. As a result, a compromised middle-tier service potentially makes it easier for an attacker to gain broad access to back-end resources.

Flowing Identity

Distributed applications can be divided into multiple secure subsystems. For example, a front-end Web application, a middle-tier Web service, a remote component, and a database represent four different security subsystems. Each performs authentication and authorization.

You must identify those subsystems that must flow the caller's identity (and associated security context) to the next downstream subsystem in order to support authorization against the original caller.

Application vs. Operating System Identity Flow

Strategies for flowing identities include using the delegation features of the operating system or passing tickets and/or credentials at the application level. For example:

- To flow identity at the application level, you typically pass credentials (or tickets) using method arguments or stored procedure parameters.

  Note **GenericPrincipal** objects that carry the authenticated caller's identity do not automatically flow across processes. This requires custom code.

  You can pass parameters to stored procedures that allow you to retrieve and process user-specific data. For example:

  ```
  SELECT CreditLimit From Table Where UserName="Bob"
  ```

  This approach is sometimes referred to as a *trusted query parameter* approach.

- Operating system identity flow requires an extended form of impersonation called delegation.

Impersonation and Delegation

Under typical circumstances, threads within a server application run using the security context of the server process. The attributes that comprise the process' security context are maintained by the process' logon session and are exposed by the process level Windows access token. All local and remote resource access is performed using the process level security context that is determined by the Windows account used to run the server process.

Impersonation

When a server application is configured for impersonation, an impersonation token is attached to the thread used to process a request. The impersonation token represents the security context of the authenticated caller (or anonymous user). Any local resource access is performed using the thread impersonation token that results in the use of the caller's security context.

Delegation

If the server application thread attempts to access a remote resource, delegation is required. Specifically, the impersonated caller's token must have network credentials. If it doesn't, all remote resource access is performed as the anonymous user (AUTHORITY\ANONYMOUS LOGON).

There are a number of factors that determine whether or not a security context can be delegated. Table 3.1 shows the various IIS authentication types and for each one indicates whether or not the security context of the authenticated caller can be delegated.

Table 3.1. IIS Authentication types

Authentication Type	Can Delegate	Notes
Anonymous	Depends	If the anonymous account (by default IUSR_MACHINE) is configured in IIS as a local account, it cannot be delegated unless the local (Web server) and remote computer have identical local accounts (with matching usernames and passwords). If the anonymous account is a domain account it can be delegated.
Basic	Yes	If Basic authentication is used with local accounts, it can be delegated if the local accounts on the local and remote computers are identical. Domain accounts can also be delegated.
Digest	No	

Integrated Windows	Depends	Integrated Windows authentication either results in NTLM or Kerberos (depending upon the version of operating system on client and server computer). NTLM does not support delegation. Kerberos supports delegation with a suitably configured environment. For more information, see How To: Implement Kerberos Delegation for Windows 2000 in the References section of this guide.
Client Certificates	Depends	Can be delegated if used with IIS certificate mapping and the certificate is mapped to a local account that is duplicated on the remote computer or is mapped to a domain account. This works because the credentials for the mapped account are stored on the local server and are used to create an Interactive logon session (which has network credentials). Active Directory certificate mapping does not support delegation.

Important Kerberos delegation under Windows 2000 is unconstrained. In other words, a user may be able to make multiple network hops across multiple remote computers. To close this potential security risk, you should limit the scope of the domain account's reach by removing the account from the Domain Users group and allow the account to be used only to log on to specific computers.

Role-Based Authorization

Most .NET Web applications will use a role-based approach to authorization. You need to consider the various role types and choose the one(s) most appropriate for your application scenario. You have the following options:

- .NET roles
- Enterprise Services (COM+) roles
- SQL Server User Defined Database roles
- SQL Server Application roles

.NET Roles

.NET roles are extremely flexible and revolve around **IPrincipal** objects that contain the list of roles that an authenticated identity belongs to. .NET roles can be used within Web applications, Web services, or remote components hosted within ASP.NET (and accessed using the HttpChannel).

You can perform authorization using .NET roles either declaratively using **PrincipalPermission** demands or programmatically in code, using imperative **PrincipalPermission** demands or the **IPrincipal.IsInRole** method.

.NET roles with Windows authentication

If your application uses Windows authentication, ASP.NET automatically constructs a **WindowsPrincipal** that is attached to the context of the current Web request (using **HttpContext.User**). After the authentication process is complete and ASP.NET has attached to object to the current request, it is used for all subsequent .NET role-based authorization.

The Windows group membership of the authenticated caller is used to determine the set of roles. With Windows authentication, .NET roles are the same as Windows groups.

.NET roles with non-Windows authentication

If your application uses a non-Windows authentication mechanism such as Forms or Passport, you must write code to create a **GenericPrincipal** object (or a custom **IPrincipal** object) and populate it with a set of roles obtained from a custom authentication data store such as a SQL Server database.

Custom IPrincipal objects

The .NET Role-based security mechanism is extensible. You can develop your own classes that implement **IPrincipal** and **IIdentity** and provide your own extended role-based authorization functionality.

As long as the custom **IPrincipal** object (containing roles obtained from a custom data store) is attached to the current request context (using **HttpContext.User**), basic role-checking functionality is ensured.

By implementing the **IPrincipal** interface, you ensure that both the declarative and imperative forms of **PrincipalPermission** demands work with your custom identity. Furthermore, you can implement extended role semantics; for example, by providing an additional method such as **IsInMultipleRoles(string [] roles)** which would allow you to test and assert for membership of multiple roles.

More information

- For more information about .NET role-based authorization, see Chapter 8, ASP.NET Security.
- For more information about creating **GenericPrincipal** objects, see How To: Create GenericPrincipal Objects with Forms Authentication in the Reference section of this guide.

Enterprise Services (COM+) Roles

Using Enterprise Services (COM+) roles pushes access checks to the middle tier and allows you to use database connection pooling when connecting to back-end databases. However, for meaningful Enterprise Services (COM+) role-based authorization, your front-end Web application must impersonate and flow the original caller's identity (using a Windows access token) to the Enterprise Services application. To achieve this, the following entries must be placed in the Web application's Web.config file.

```
<authentication mode="Windows" />

<identity impersonate="true" />
```

If it is sufficient to use declarative checks at the method level (to determine which users can call which methods), you can deploy your application and update role membership using the Component Services administration tool.

If you require programmatic checks in method code, you lose some of the administrative and deployment advantages of Enterprise Services (COM+) roles, because role logic is hard-coded.

SQL Server User Defined Database Roles

With this approach, you create roles in the database, assign permissions based on the roles and map Windows group and user accounts to the roles. This approach requires you to flow the caller's identity to the back end (if you are using the preferred Windows authentication to SQL Server).

SQL Server Application Roles

With this approach, permissions are granted to the roles within the database, but SQL Server application roles contain no user or group accounts. As a result, you lose the granularity of the original caller.

With application roles, you are authorizing access to a specific application (as opposed to a set of users). The application activates the role using a built-in stored procedure that accepts a role name and password. One of the main disadvantages of this approach is that it requires the application to securely manage credentials (the role name and associated password).

More information

For more information about SQL Server user defined database roles and application roles, see Chapter 12, Data Access Security.

.NET Roles versus Enterprise Services (COM+) Roles

The following table presents a comparison of the features of .NET roles and Enterprise Services (COM+) roles.

Table 3.2. Comparing Enterprise Services roles with .NET roles

Feature	Enterprise Services Roles	.NET Roles
Administration	Component Services Administration Tool	Custom
Data Store	COM+ Catalog	Custom data store (for example, SQL Server or Active Directory)
Declarative	Yes [SecurityRole("Manager")]	Yes [PrincipalPermission(SecurityAction.Demand, Role="Manager")]
Imperative	Yes ContextUtil.IsCallerInRole()	Yes IPrincipal.IsInRole
Class, Interface and Method Level Granularity	Yes	Yes
Extensible	No	Yes (using custom IPrincipal implementation)
Available to all .NET components	Only for components that derive from ServicedComponent base class	Yes
Role Membership	Roles contain Windows group or user accounts	When using WindowsPrincipals, roles ARE Windows groups–no extra level of abstraction
Requires explicit Interface implementation	Yes To obtain method level authorization, an interface must be explicitly defined and implemented	No

Using .NET Roles

You can secure the following items with .NET roles:

- Files
- Folders
- Web pages (.aspx files)
- Web services (.asmx files)
- Objects
- Methods and properties
- Code blocks within methods

The fact that you can use .NET roles to protect operations (performed by methods and properties) and specific code blocks means that you can protect access to local and remote resources accessed by your application.

Note The first four items in the preceding list (Files, folders, Web pages, and Web services) are protected using the **UrlAuthorizationModule**, which can use the role membership of the caller (and the caller's identity) to make authorization decisions.

If you use Windows authentication, much of the work required to use .NET roles is done for you. ASP.NET constructs a **WindowsPrincipal** object and the Windows group membership of the user determines the associated role set.

To use .NET roles with a non-Windows authentication mechanism, you must write code to:

- Capture the user's credentials.
- Validate the user's credentials against a custom data store such as a SQL Server database.
- Retrieve a role list, construct a **GenericPrincipal** object and associate it with the current Web request.

The **GenericPrincipal** object represents the authenticated user and is used for subsequent .NET role checks, such as declarative **PrincipalPermission** demands and programmatic **IPrincipal.IsInRole** checks.

More information

For more information about the process involved in creating a **GenericPrincipal** object for Forms authentication, see Chapter 8, ASP.NET Security.

Checking role membership

The following types of .NET role checks are available:

Important .NET role checking relies upon an **IPrincipal** object (representing the authenticated user) being associated with the current request. For ASP.NET Web applications, the **IPrincipal** object must be attached to **HttpContext.User**. For Windows Forms applications, the **IPrincipal** object must be attached to **Thread.CurrentPrincipal**.

- **Manual role checks**. For fine-grained authorization, you can call the **IPrincipal.IsInRole** method to authorize access to specific code blocks based on the role membership of the caller. Both AND and OR logic can be used when checking role membership.
- **Declarative role checks (gates to your methods)**. You can annotate methods with the **PrincipalPermissionAttribute** class (which can be shortened to **PrincipalPermission**), to declaratively demand role membership. These support OR logic only. For example you can demand that a caller is in at least one specific role (for example, the caller must be a teller or a manager). You cannot specify that a caller must be a manager and a teller using declarative checks.
- **Imperative role checks (checks within your methods)**. You can call **PrincipalPermission.Demand** within code to perform fine-grained authorization logic. Logical AND and OR operations are supported.

Role-checking examples

The following code fragments show some example role checks using programmatic, declarative, and imperative techniques.

1. Authorizing Bob to perform an operation:

Note Although you can authorize individual users, you should generally authorize based on role membership, which allows you to authorize sets of users who share the same privileges within your application.

- Direct user name check

```
GenericIdentity userIdentity = new GenericIdentity("Bob");
if (userIdentity.Name=="Bob")
{
}
```

- Declarative check

```
[PrincipalPermissionAttribute(SecurityAction.Demand,
  User="Bob")]
```

```
        public void DoPrivilegedMethod()
        {
        }
```

- Imperative check

```
        PrincipalPermission permCheckUser = new
          PrincipalPermission(
                                                  "Bob",
                                                    null);
        permCheckUser.Demand();
```

2. Authorizing tellers to perform an operation:

- Direct role name check

```
        GenericIdentity userIdentity = new GenericIdentity("Bob");
        // Role names would be retrieved from a custom data store
        string[] roles = new String[]{"Manager", "Teller"};
        GenericPrincipal userPrincipal = new
          GenericPrincipal(userIdentity,
                                                      roles);
        if (userPrincipal.IsInRole("Teller"))
        {
        }
```

- Declarative check

```
        [PrincipalPermissionAttribute(SecurityAction.Demand,
    Role="Teller")]
        void SomeTellerOnlyMethod()
        {
        }
```

- Imperative check

```
public SomeMethod()
{
  PrincipalPermission permCheck = new PrincipalPermission(
                                          null,"Teller");
  permCheck.Demand();
  // Only Tellers can execute the following code
  // Non members of the Teller role result in a security
    exception
  . . .
}
```

3. Authorize managers OR tellers to perform operation:

- Direct role name check

```
if (Thread.CurrentPrincipal.IsInRole("Teller") ||
    Thread.CurrentPrincipal.IsInRole("Manager"))
{
  // Perform privileged operations
}
```

- Declarative check

```
[PrincipalPermissionAttribute(SecurityAction.Demand,
  Role="Teller"),
 PrincipalPermissionAttribute(SecurityAction.Demand,
   Role="Manager")]
public void DoPrivilegedMethod()
{
Ã‚Â…
```

```
        }
```

- Imperative check

```
        PrincipalPermission permCheckTellers = new
          PrincipalPermission(
                                                null,"Teller");
        PrincipalPermission permCheckManagers = new
          PrincipalPermission(
                                               null,"Manager");
      (permCheckTellers.Union(permCheckManagers)).Demand();
```

4. Authorize only those people who are managers AND tellers to perform operation:

- Direct role name check

```
        if (Thread.CurrentPrincipal.IsInRole("Teller") &&
            Thread.CurrentPrincipal.IsInRole("Manager"))
        {
          // Perform privileged operation
        }
```

- Declarative check

It is not possible to perform AND checks with .NET roles declaratively. Stacking **PrincipalPermission** demands together results in a logical OR.

- Imperative check

```
        PrincipalPermission permCheckTellers = new
          PrincipalPermission(
                                               null,"Teller");
        permCheckTellers.Demand();
        PrincipalPermission permCheckManagers = new
          PrincipalPermission(
                                             null, "Manager");
```

```
permCheckManagers.Demand();
```

Choosing an Authentication Mechanism

This section presents guidance that is designed to help you choose an appropriate authentication mechanism for common application scenarios. You should start by considering the following issues:

- **Identities**. A Windows authentication mechanism is appropriate only if your application's users have Windows accounts that can be authenticated by a trusted authority accessible by your application's Web server.
- **Credential management**. One of the key advantages of Windows authentication is that it enables you to let the operating system take care of credential management. With non-Windows approaches, such as Forms authentication, you must carefully consider where and how you store user credentials. The two most common approaches are to use:
  - SQL Server databases
  - User objects within Active Directory

  For more information about the security considerations of using SQL Server as a credential store, see Chapter 12, Data Access Security.

  For more information about using Forms authentication against custom data stores (including Active Directory), see Chapter 8, ASP.NET Security.
- **Identity flow**. Do you need to implement an impersonation/delegation model and flow the original caller's security context at the operating system level across tiers? For example, to support auditing or per-user (granular) authorization. If so, you need to be able to impersonate the caller and delegate their security context to the next downstream subsystem, as described in the "Delegation" section earlier in this chapter.
- **Browser type**. Do your users all have Internet Explorer or do you need to support a user base with mixed browser types? Table 3.3 illustrates which authentication mechanisms require Internet Explorer browsers, and which support a variety of common browser types.

  Table 3.3. Authentication browser requirements

Authentication Type	Requires Internet Explorer	Notes
Forms	No	
Passport	No	
Integrated Windows (Kerberos or NTLM)	Yes	Kerberos also requires Windows 2000 or later operating systems on the client and server computers and accounts configured for delegation. For more information, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.
Basic	No	Basic authentication is part of the HTTP 1.1 protocol that is supported by virtually all browsers
Digest	Yes	
Certificate	No	Clients require X.509 certificates

Internet Scenarios

- The basic assumptions for Internet scenarios are:
  - Users do not have Windows accounts in the server's domain or in a trusted domain accessible by the server.

- Users do not have client certificates.

Figure 3.4 shows a decision tree for choosing an authentication mechanism for Internet scenarios.

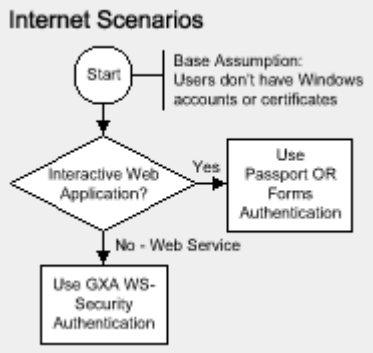

Figure 3.4. Choosing an authentication mechanism for Internet applications

For more information about Web service security and the WS-Security specification, part of the Global XML Architecture (GXA) initiative, see Chapter 10, Web Services Security.

Forms / Passport comparison

This section summarizes the relative merits of Forms and Passport authentication.

Advantages of Forms authentication

- Supports authentication against a custom data store; typically a SQL Server database or Active Directory.
- Supports role-based authorization with role lookup from a data store.
- Smooth integration with Web user interface.
- ASP.NET provides much of the infrastructure. Relatively little custom code is required in comparison to classic ASP.

Advantages of Passport authentication

- Passport is a centralized solution.
- It removes credential management issues from the application.
- It can be used with role-based authorization schemes.
- It is very secure as it is built on cryptography technologies.

More information

- For more information about Web service authentication approaches, see Chapter 10, Web Services Security.
- For more information about using Forms Authentication with SQL Server, see How To: Use Forms authentication with SQL Server 2000 in the Reference section of this guide.

Intranet / Extranet Scenarios

Figure 3.5 shows a decision tree that can be used to help choose an authentication mechanism for intranet and extranet application scenarios.

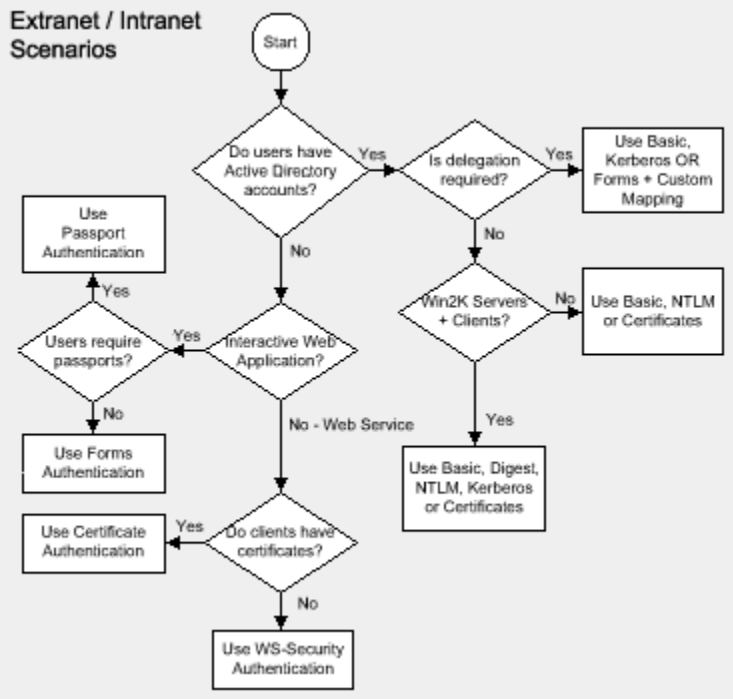

Figure 3.5. Choosing an authentication mechanism for intranet and extranet applications

Authentication Mechanism Comparison

The following table presents a comparison of the available authentication mechanisms.

Table 3.4: Available authentication methods

	Basic	Digest	NTLM	Kerberos	Certs	Forms	Passport
Users need Windows accounts in server's domain	Yes	Yes	Yes	Yes	No	No	No
Supports delegation*	Yes	No	No	Yes	Can do	Yes	Yes
Requires Win2K clients and servers	No	Yes	No	Yes	No	No	No
Credentials passed as clear text (requires SSL)	Yes	No	No	No	No	Yes	No
Supports non-IE browsers	Yes	No	No	No	Yes	Yes	Yes

* Refer to the "Delegation" topic in the "Flowing Identity" section earlier in this chapter for details.

Summary

- Designing distributed application authentication and authorization approaches is a challenging task. Proper authentication and authorization design during the early design phases of your application development helps mitigate many of the top security risks. The following summarizes the information in this chapter:

- Use the trusted subsystem resource access model to gain the benefits of database connection pooling.
- If your application does not use Windows authentication, use .NET role checking to provide authorization. Validate credentials against a custom data store, retrieve a role list and create a **GenericPrincipal** object. Associate it with the current Web request (**HttpContext.User**).
- If your application uses Windows authentication and doesn't use Enterprise Services, use .NET roles. Remember that for Windows authentication, .NET roles are Windows groups.
- If your application uses Windows authentication and Enterprise Services, consider using Enterprise Services (COM+) roles.
- For meaningful role-based authorization using Enterprise Services (COM+) roles, the original caller's identity must flow to the Enterprise Services application. If the Enterprise Services application is called from an ASP.NET Web application, this means that the Web application must use Windows authentication and be configured for impersonation.
- Annotate methods with the **PrincipalPermission** attribute to declaratively demand role membership. The method is not called if the caller is not in the specified role and a security exception is generated.
- Call **PrincipalPermission.Demand** within method code (or use **IPrincipal.IsInRole**) for fine-grained authorization decisions.
- Consider implementing a custom **IPrincipal** object to gain additional role-checking semantics.

Secure Communication

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter introduces the two core technologies that can be used to provide message confidentiality and message integrity for data that flows across the network between clients and servers on the Internet and corporate intranet. These are SSL and IPSec. It also discusses RPC encryption, which can be used to secure the communication with remote serviced components. (10 printed pages)

Contents

Many applications pass security sensitive data across networks to and from end users and between intermediate application nodes. Sensitive data might include credentials used for authentication, or data such as credit card numbers or bank transaction details. To guard against unwanted information disclosure and to protect the data from unauthorized modification while in transit, the channel between communication end points must be secured.

Secure communication provides the following two features:

- **Privacy**. Privacy is concerned with ensuring that data remains private and confidential, and cannot be viewed by eavesdroppers who may be armed with network monitoring software. Privacy is usually provided by means of encryption.
- **Integrity**. Secure communication channels must also ensure that data is protected from accidental or deliberate (malicious) modification while in transit. Integrity is usually provided by using Message Authentication Codes (MACs).

This chapter covers the following secure communication technologies:

- **Secure Sockets Layer / Transport Layer Security (SSL/TLS)**. This is most commonly used to secure the channel between a browser and Web server. However, it can also be used to secure Web service messages and communications to and from a database server running Microsoft® SQL Server™ 2000.
- **Internet Protocol Security (IPSec)**. IPSec provides a transport level secure communication solution and can be used to secure the data sent between two computers; for example, an application server and a database server.
- **Remote Procedure Call (RPC) Encryption**. The RPC protocol used by Distributed COM (DCOM) provides an authentication level (packet privacy) that results in the encryption of every packet of data sent between client and server.

Know What to Secure

When a Web request flows across the physical deployment tiers of your application, it crosses a number of communication channels. A commonly used Web application deployment model is shown in Figure 4.1.

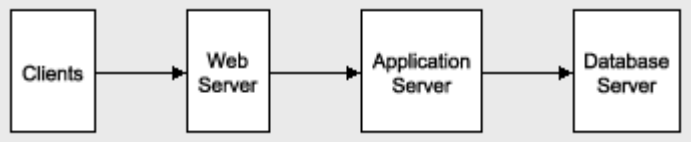

Figure 4.1. A typical Web deployment model

In this typical deployment model, a request passes through three distinct channels. The client-to-Web server link may be over the Internet or corporate intranet and typically uses HTTP. The remaining two links are between internal servers within your corporate domain. Nonetheless, all three links represent potential security concerns. Many purely intranet-based applications convey security sensitive data between tiers; for example, HR and payroll applications that deal with sensitive employee data.

Figure 4.2 shows how each channel can be secured by using a combination of SSL, IPSec and RPC encryption.

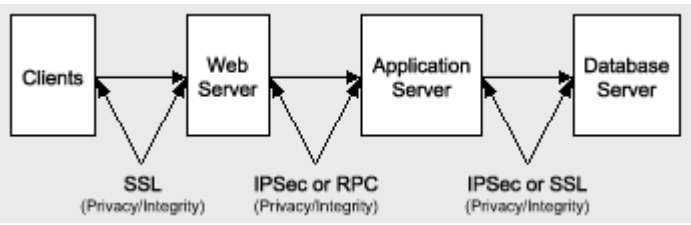

Figure 4.2. A typical Web deployment model, with secure communications

The choice of technology depends on a number of factors including the transport protocol, end point technologies, and environmental considerations (such as hardware, operating system versions, firewalls, and so on).

SSL / TLS

SSL/TLS is used to establish an encrypted communication channel between client and server. The handshake mechanism used to establish the secure channel is well documented and details can be found in the following articles in the Microsoft Knowledge Base:

- Q257591, Description of the Secure Sockets Layer (SSL) Handshake
- Q257587, Description of the Server Authentication Process During the SSL Handshake
- Q257586, Description of the Client Authentication Process During the SSL Handshake

Using SSL

When you use SSL you should be aware of the following:

- When SSL is applied, the client uses the HTTPS protocol (and specifies an https:// URL) and the server listens on TCP port 443.
- You should monitor your application's performance when you enable SSL.

  SSL uses complex cryptographic functions to encrypt and decrypt data and as a result impacts the performance of your application. The largest performance hit occurs during the initial handshake, where asymmetric public/private-key encryption is used. Subsequently (after a secure session key is generated and exchanged), faster, symmetric encryption is used to encrypt application data.

- You should optimize pages that use SSL by including less text and simple graphics in those pages.
- Because the performance hit associated with SSL is greatest during session establishment, ensure that your connections do not time out.

You can fine tune this by increasing the value of the **ServerCacheTime** registry entry. For more information, see article Q247658, HOW TO: Configure Secure Sockets Layer Server and Client Cache Elements in the Microsoft Knowledge Base.

- SSL requires a server authentication certificate to be installed on the Web server (or database server if you are using SSL to communicate with SQL Server 2000). For more information about installing server authentication certificates, see How To: Set Up SSL on a Web Server in the How To section of this guide.

IPSec

IPSec can be used to secure the data sent between two computers; for example, an application server and a database server. IPSec is completely transparent to applications as encryption, integrity, and authentication services are implemented at the transport level. Applications continue to communicate with one another in the normal manner using TCP and UDP ports.

Using IPSec you can:

- Provide message confidentiality by encrypting all of the data sent between two computers.
- Provide message integrity between two computers (without encrypting data).
- Provide mutual authentication between two computers (not users). For example, you can help secure a database server by establishing a policy that permits requests only from a specific client computer (for example, an application or Web server).
- Restrict which computers can communicate with one another. You can also restrict communication to specific IP protocols and TCP/UDP ports.

Note IPSec is not intended as a replacement for application level security. Today it is used as a defense-in-depth mechanism or to secure insecure applications without changing them, and to secure non-TLS protocols from network-wire attacks.

Using IPSec

When you use IPSec you should be aware of the following:

- IPSec can be used for both authentication and encryption.
- There are no IPSec APIs for developers to programmatically control settings. IPSec is completely controlled and configured through the IPSec snap-in, within the Local Security Policy Microsoft Management Console (MMC).
- IPSec in the Microsoft Windows® 2000 operating system cannot secure all types of IP traffic.

  Specifically, it cannot be used to secure Broadcast, Multicast, Internet Key Exchange, or Kerberos (which is already a secure protocol) traffic.

  For more information, see article Q253169, Traffic That Can and Cannot Be Secured by IPSec, in the Microsoft Knowledge Base.

- You use IPSec filters to control when IPSec is applied.

  To test the IPSec policies, use IPSec Monitor. IPSec Monitor (Ipsecmon.exe) provides information about which IPSec policy is active and whether a secure channel between computers is established.

  For more information, see the Knowledge Base articles:

  - Q313195, HOW TO: Use IPSec Monitor in Windows 2000
  - Q231587, Using the IP Security Monitor Tool to View IPSec Communications

- To establish a trust between two servers, you can use IPSec with mutual authentication. This uses certificates to authenticate both computers.

  For more information, see the following Knowledge Base articles:

- Q248711, Mutual Authentication Methods Supported for L2TP/IPSec
- Q253498, HOW TO: Install a Certificate for Use with IP Security

- If you need to use IPSec to secure communication between two computers that are separated by a firewall, make sure that the firewall does not use Network Address Translation (NAT). IPSec does not work with any NAT-based devices.

  For more information and configuration steps, see article Q233256, HOW TO Enable IPSec Traffic through a Firewall in the Microsoft Knowledge Base and How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

RPC Encryption

RPC is the underlying transport mechanism used by DCOM. RPC provides a set of configurable authentication levels that range from no authentication (and no protection of data) to full encryption of parameter state.

The most secure level (RPC Packet Privacy) encrypts parameter state for every remote procedure call (and therefore every DCOM method invocation). The level of RPC encryption, 40-bit or 128-bit, depends on the version of the Windows operating system that is running on the client and server computers.

Using RPC Encryption

You are most likely to want to use RPC encryption when your Web-based application communicates with serviced components (within Enterprise Services server applications) located on remote computers.

In this event, to use RPC Packet Privacy authentication (and encryption) you must configure both the client and the server. A process of high-water mark negotiation occurs between client and server, which ensures that the higher of the two (client and server) settings are used.

The server settings can be defined at the (Enterprise Services) application level, either by using .NET attributes within your serviced component assembly, or by using the Component Services administration tool at deployment time.

If the client is an ASP.NET Web application or Web service, the authentication level used by the client is configured using the **comAuthenticationLevel** attribute on the <**processModel**> element within Machine.config. This provides the default authentication level for all ASP.NET applications that run on the Web server.

More information

For more information about RPC authentication level negotiation and service component configuration, see Chapter 9, Enterprise Services Security.

Point-to-Point Security

- Point-to-point communication scenarios can be broadly categorized into the following topics:
  - Browser to Web Server
  - Web Server to Remote Application Server
  - Application Server to Database Server

Browser to Web Server

To secure sensitive data sent between a browser and Web server, use SSL. You need to use SSL in the following situations:

- You are using Forms authentication and need to secure the clear text credentials submitted to a Web server from a logon form.

  In this scenario, you should use SSL to secure access to all pages (not just the logon page) to ensure that the authentication cookie, generated as a result on the initial authentication process, remains secure throughout the lifetime of the client's browser session with the application.

- You are using Basic authentication and need to secure the (Base64 encoded) clear text credentials.

  You should use SSL to secure access to all pages (not just the initial log on), as Basic authentication sends the clear text credentials to the Web server with all requests to the application (not just the initial one).

  > **Note** Base64 is used to encode binary data as printable ASCII text. Unlike encryption, it does not provide message integrity or privacy.

- Your application passes sensitive data between the browser and Web server (and vice-versa); for example, credit card numbers or bank account details.

Web Server to Remote Application Server

The transport channel between a Web server and a remote application server should be secured by using IPSec, SSL or RPC Encryption. The choice depends on the transport protocols, environmental factors (operating system versions, firewalls and so on).

- **Enterprise Services**. If your remote server hosts one or more serviced components (in an Enterprise Services server application) and you are communicating directly with them (and as a result using DCOM), use RPC Packet Privacy encryption.

  For more information about how to configure RPC encryption between a Web application and remote serviced component, see Chapter 9,Enterprise Services Security.

- **Web Services**. If your remote server hosts a Web service, you can choose between IPSec and SSL.

  You should generally use SSL because the Web service already uses the HTTP transport. SSL also allows you to only encrypt the data sent to and from the Web service (and not all traffic sent between the two computers). IPSec results in the encryption of all traffic sent between the two computers.

  > **Note** Message-level security (including data encryption) is addressed by the Global XML Web Services Architecture (GXA) initiative and specifically the WS-Security specification. Microsoft provides the Web Services Development Toolkit to allow you to develop message level security solutions.

- **.NET Components (using .NET Remoting)**. If your remote server hosts one or more .NET components and you connect to them over the TCP channel, you can use IPSec to provide a secure communication link. If you host the .NET components within ASP.NET, you can use SSL (configured using IIS).

Application Server to Database Server

To secure the data sent between an application server and database server, you can use IPSec. If your database server runs SQL Server 2000 (and the SQL Server 2000 network libraries are installed on the application server), you can use SSL. This latter option requires a server authentication certificate to be installed in the database server's machine store.

You may need to secure the link to the database server in the following situations:

- You are connecting to the database server and are not using Windows authentication. For example, you may be using SQL authentication to SQL Server or you may be connecting to a non-SQL Server database. In these cases, the credentials are passed in clear text, which can represent a significant security concern.

  Note One of the key benefits of using Windows authentication to SQL Server is that it means that the credentials are never passed across the network. For more information about Windows and SQL authentication, see Chapter 12, Data Access Security.

- Your application may be submitting and retrieving sensitive data to and from the database (for example, payroll data).

Using SSL to SQL Server

Consider the following points if you use SSL to secure the channel to a SQL Server database:

- For SSL to work, you must install a server authentication certificate in the machine store on the database server computer. The client computer must also have a root Certificate Authority certificate from the same (or trusting) authority that issued the server certificate.
- Clients must have the SQL Server 2000 connectivity libraries installed. Earlier versions or generic libraries will not work.
- SSL only works for TCP/IP (the recommended communication protocol for SQL Server) and named pipes.
- You can configure the server to force the use of encryption for all connections (from all clients).
- On the client, you can:
  - Force the use of encryption for all outgoing connections.
  - Allow client applications to choose whether or not to use encryption on a per-connection basis, by using the connection string.
- Unlike IPSec, configuration changes are not required if the client or server IP addresses change.

More information

For more information about using SSL to SQL Server, see the following resources:

- How To: Use SSL to Secure Communication with SQL Server 2000 in the Reference section of this guide
- Webcast: Microsoft SQL Server 2000: How to Configure SSL Encryption (April 23, 2002)

Choosing Between IPSec and SSL

Consider the following points when choosing between IPSec and SSL:

- IPSec can used to secure all IP traffic between computers; SSL is specific to an individual application.
- IPSec is a computer-wide setting and does not support the encryption of specific network connections. However, sites can be partitioned to use or not use SSL. Also, when you use SSL to connect to SQL Server, you can choose on a per connection basis (from the client application) whether or not to use SSL.
- IPSec is transparent to applications, so it can be used with secure protocols that run on top of IP such as HTTP, FTP and SMTP. However, SSL/TLS is closely tied to the application.
- IPSec can be used for computer authentication in addition to encryption. This is particularly significant for trusted subsystem scenarios, where the database authorizes a fixed identity from a specific application (running on a specific computer). IPSec can be used to ensure that only the specific application server can connect to the database server, in order to prevent attacks from other computers.
- IPSec requires that both computers run Windows 2000 or later.
- SSL can work through a NAT-based firewall; IPSec cannot.

Farming and Load Balancing

If you use SSL in conjunction with multiple virtual Web sites, you need to use unique IP addresses or unique port numbers. You cannot use multiple sites with the same IP address and port number. If the IP address is combined with a server affinity setting in a load balancer, this will work fine.

More Information

For more information, see Q187504, HTTP 1.1 Host Headers Are Not Supported When You Use SSL, in the Microsoft Knowledge Base.

Summary

This chapter described how a combination of SSL, IPSec and RPC encryption can be used to provide an end-to-end secure communication solution for your distributed application. To summarize:

- Channel security is a concern for data passed over the Internet and on the corporate intranet.
- Consider the security requirements of the Web browser to Web server, Web server to application server, and application server to database server links.
- Secure communication provides privacy and integrity. It does not protect you from non-repudiation (for this use, client certificates)
- Channel security options include SSL, IPSec and RPC Encryption. The latter option applies when your application uses DCOM to communicate with remote serviced components.
- If you use SSL to communicate with SQL Server, the application can choose (on a per-connection basis) whether or not to encrypt the connection.
- IPSec encrypts all IP traffic that flows between two computers.
- The choice of security mechanism is dependent upon transport protocol, operating system versions, and network considerations (including firewalls).
- There is always a trade-off between secure communication and performance. Choose the level of security that is appropriate to your application requirements.

Intranet Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
- Microsoft® ASP.NET
- Microsoft SQL Server™

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter describes how to secure common intranet application scenarios. It presents the characteristics of each scenario and describes the steps necessary to secure the scenario. Analysis sections are also included to provide further information. (34 printed pages)

Contents

Access to intranet applications is restricted to a limited group of authorized users (such as employees that belong to a domain). While an intranet setting limits the exposure of your application, you may still face several challenges when you develop authentication, authorization, and secure communication strategies. For example, you may have non-trusting domains, which make it difficult to flow a caller's security context and identity through to the back-end resources within your system. You may also be operating within a heterogeneous environment with mixed browser types. This makes it more difficult to use a common authentication mechanism.

If you have a homogenous intranet where all computers run the Microsoft® Windows® 2000 operating system or later and you have a domain where users are trusted for delegation, delegation of the original caller's security context to the back end becomes an option.

You must also consider secure communication. Despite the fact that your application runs in an intranet environment, you cannot consider the data sent over the network secure. It is likely that you will need to secure the data sent between browsers and the Web server in addition to data sent between application servers and databases.

The following common intranet scenarios are used in this chapter to illustrate key authentication, authorization, and secure communication techniques:

- ASP.NET to Microsoft SQL Server™
- ASP.NET to Enterprise Services to SQL Server
- ASP.NET to Web Services to SQL Server
- ASP.NET to Remoting to SQL Server

In addition, this chapter describes a Windows 2000 delegation scenario (Flowing the Original Caller to the Database), in which the original caller's security context and identity flows at the operating system level from browser to database using intermediate Web and application servers.

Note Several scenarios described in this chapter either replace the default ASPNET account used to run ASP.NET applications or change its password to allow duplicated accounts to be created on remote computers. These scenarios update the <**processModel**> element of Machine.config. This results in credentials being stored in clear text within machine.config. For a detailed discussion of this topic, see Accessing Network Resources in Chapter 8, "ASP.NET Security."

ASP.NET to SQL Server

In this scenario, a HR database serves per-user data securely on a homogenous intranet. The application uses a trusted subsystem model and executes calls on behalf of the original callers. The application authenticates callers by using Integrated Windows authentication and makes calls to the database using the ASP.NET process identity. Due to the sensitive nature of the data, SSL is used between the Web server and clients.

The basic model for this application scenario is shown in Figure 5.1.

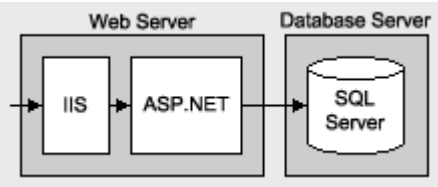

Figure 5.1. ASP.NET to SQL Server

Characteristics

This scenario has the following characteristics:

- Clients have Internet Explorer.
- User accounts are in Microsoft Active Directory® directory service.
- The application provides sensitive, per-user data.
- Only authenticated clients should access the application.
- The database trusts the application to authenticate users properly (that is, the application makes calls to the database on behalf of the users).
- Microsoft SQL Server is using a single database user role for authorization.

Secure the Scenario

In this scenario, the Web server authenticates the caller and restricts access to local resources by using the caller's identity. You don't have to impersonate within the Web application in order to restrict access to resources against the original caller. The database authenticates against the ASP.NET default process identity, which is a least privileged account (that is, the database trusts the ASP.NET application).

Table 5.1. Security measures

Category	Details
Authentication	Provide strong authentication at the Web server to authenticate original callers by using Integrated Windows authentication in IIS. Use Windows authentication within ASP.NET (no impersonation). Secure connections to the database using SQL Server configured for Windows authentication. The database trusts the ASP.NET worker process to make calls. Authenticate the ASP.NET process identity at the database.
Authorization	Configure resources on the Web server using ACLs tied to the original callers. For easier administration, users are added to Windows groups and groups are used within the ACLs. The Web application performs .NET role checks against the original caller to restrict access to pages.
Secure Communication	Secure sensitive data sent between the Web server and the database Secure sensitive data sent between the original callers and the Web application

The Result

Figure 5.2 shows the recommended security configuration for this scenario.

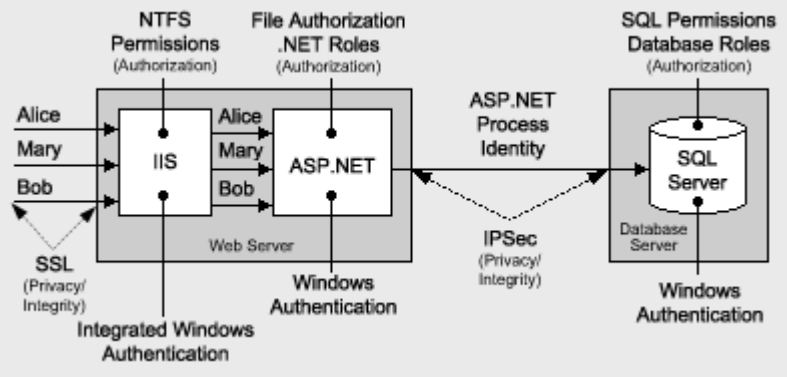

Figure 5.2. The recommended security configuration for the ASP.NET to SQL Server intranet scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Creating custom ASP.NET accounts (see How To: Create a Custom Account to Run ASP.NET in the Reference section of this guide)
- Creating a least privileged database account (see Chapter 12, Data Access Security)
- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)

Configuring IIS

Step	More Information
Disable Anonymous access for your Web application's virtual root directory Enable Integrated Windows Authentication	To work with IIS authentication settings, use the IIS MMC snap-in. Right-click your application's virtual directory, and then click **Properties**. Click the **Directory Security** tab, and then click **Edit** within the **Anonymous access and authentication control** group.

Configuring ASP.NET

Step	More Information

Change the ASPNET password to a known strong password value	ASPNET is a least privileged local account used by default to run ASP.NET Web applications.
	Set the ASPNET account's password to a known value by using Local Users and Groups.
	Edit Machine.config located in %windir%\Microsoft.NET\Framework\ v1.0.3705\CONFIG
	and reconfigure the password attribute on the <**processModel**> element Default
	
	`<!-- userName="machine" password="AutoGenerate"`
	
	`-- >`

	Becomes
	
	`<!-- userName="machine"`
	
	`password="YourNewStrongPassword" --  >`

Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your application's virtual directory root Set the <**authentication**> element to: `<authentication mode="Windows" />`
Make sure impersonation is off	Impersonation is off by default; however, double check to ensure that it's turned off in Web.config, as follows: `<identity impersonate="false" />` The same effect can be achieved by removing the <**identity**> element.

Configuring SQL Server

Step	More Information
Create a Windows account on your SQL Server computer that matches the ASP.NET process account (ASPNET)	The user name and password must match the ASPNET account. Give the account the following privileges: • Access this computer from the network • Deny logon locally • Log on as a batch job
Configure SQL Server for Windows authentication	
Create a SQL Server Login for the local ASPNET account	This grants access to the SQL Server
Create a new database user and map the login name to the database user	This grants access to the specified database
Create a new user-defined database role and add the database user to the role	
Establish database permissions for the database role	Grant minimum permissions For more information, see Chapter 12, Data Access Security.

Configuring secure communication

Step	More Information
Configure the Web site for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure IPSec between Web server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Analysis

- Integrated Windows authentication in IIS is ideal in this scenario because all users have Windows accounts and are using Microsoft Internet Explorer. The benefit of Integrated Windows authentication is that the user's password is never sent over the network. Additionally, the logon is transparent for the user because Windows uses the current interactive user's logon session.
- ASP.NET is running as least privileged account, so potential damage from compromise is mitigated.
- You don't need to impersonate in ASP.NET to perform .NET role checks or to secure resources within Windows ACLs against the original caller. To perform .NET role checks against the original caller, the **WindowsPrincipal** object that represents the original caller is retrieved from the HTTP context as follows:

```
WindowsPrincipal wp = (HttpContext.Current.User as
  WindowsPrincipal);
if ( wp.IsInRole("Manager") )
{
  // User is authorized to perform manager-specific functionality
}
```

The ASP.NET **FileAuthorizationModule** provides ACL checks against the original caller for ASP.NET file types that are mapped within IIS to the aspnet_isapi.dll. For static file types such as .jpg, .gif and .htm files, IIS acts as the gatekeeper and performs access checks using the original caller's identity, based on the NTFS permissions associated with the file.

- Using Windows authentication to SQL Server means that you avoid storing credentials in files and passing credentials over the network to the database server.
- The use of a duplicated Windows account on the database server (one that matches the ASPNET local account) results in increased administration. If a password is changed on one computer, it must be synchronized and updated on the other. In some scenarios, you may be able to use a least-privileged domain account for easier administration.
- The duplicated local account approach also works in the presence of a firewall where the ports required for Windows authentication may not be open. The use of Windows authentication and domain accounts may not work in this scenario.
- You'll need to ensure that your Windows groups are as granular as your security needs. Because .NET role-based security is based on Windows group membership this solution relies on Windows groups being set up at the correct level of granularity to match the categories of users (sharing the same security privileges) who access the application. The Windows groups that you use here to manage roles could be local to that computer or domain groups
- SQL Server database user roles are preferred to SQL server application roles to avoid the associated password management and connection pooling issues associated with the use of SQL application roles.

  Applications activate SQL application roles by calling a built-in stored procedure with a role name and a password. Therefore, the password must be stored securely. Database connection pooling must also be disabled when you use SQL application roles, which severely impacts application scalability.

  For more information about SQL Server database user roles and SQL Server application roles, see Chapter 12, Data Access Security.

- The database user is added to a database user role and permissions are assigned for the role so that if the database account changes; you don't have to change the permissions on all database objects.

Q&A

- **Why can't I enable impersonation for the Web application, so that I can secure the resources accessed by my Web application using ACLs configured against the original caller?**

  If you enable impersonation, the impersonated security context will not have network credentials (assuming delegation is not enabled and you are using Integrated Windows authentication). Therefore, the remote call to SQL Server will use a NULL session, which will result in a failed call. With impersonation disabled, the remote request will use the ASP.NET process identity.

  The preceding scenario uses the ASP.NET **FileAuthorizationModule**, which performs authorization using Windows ACLs against the original caller identity and does not require impersonation.

  If you use Basic authentication instead of Integrated Windows authentication (NTLM) and you do enable impersonation, each call to the database would use the original caller's security context. Each user account (or the Windows groups to which the user belongs) would require SQL Server logins. Permissions on database objects would need to be secured against the Windows group (or original caller).

- **The database doesn't know who the original caller is. How can I create an audit trail?**

  Audit end user activity within the Web application or pass the identity of the user explicitly as a parameter of the data access call.

Related Scenarios

Non-Internet Explorer Browsers

Integrated Windows authentication to IIS requires Internet Explorer. In a mixed browser environment, your typical options would include:

- **Basic authentication and SSL**. Basic authentication is supported by most browsers. Since the user's credentials are passed over the network, you must use SSL to secure the scenario.
- **Client certificates**. Individual client certificates can either be mapped to a unique Windows account or a single Windows account can be used to represent all clients. The use of client certificates also requires SSL.
- **Forms authentication**. Forms authentication can validate credentials against a custom data store such as a database or against Active Directory.

  If you authenticate against Active Directory, make sure that you retrieve only the necessary groups that are pertinent to your application. Just like you shouldn't issue queries against a database using SELECT * clauses, you shouldn't blindly retrieve all groups from Active Directory.

  If you authenticate against a database, you need to carefully parse the input used in SQL commands to protect against SQL injection attacks, and you should store password hashes (with salt) in the database instead of clear text or encrypted passwords.

  For more information about using SQL Server as a credential store and storing passwords in the database, see Chapter 12, Data Access Security.

Notice that in all cases, if you don't use Integrated Windows authentication, where the platform manages credentials for you, you end up using SSL. However, this benefit pertains strictly to the authentication process. If you are passing security sensitive data over the network, you must still use IPSec or SSL.

SQL authentication to the database

In some scenarios you may be forced to use SQL authentication instead of the preferred Windows authentication. For example, there may be a firewall between the Web application and database, or the Web server may not be a member of your domain for security reasons. This also prevents Windows authentication. In this case, you might use SQL authentication between the database and Web server. To secure this scenario, you should:

- Use the Data Protection API (DPAPI) to secure database connection strings that contain usernames and passwords. For more information, see the following resources:
  - Storing Database Connection Strings Securely, in Chapter 12, "Data Access Security"

- How To: Use DPAPI (Machine Store) from ASP.NET in the Reference section of this guide
- How To: Use DPAPI (User Store) from ASP.NET with Enterprise Services in the Reference section of this guide
- How To: Create a DPAPI Library in the Reference section of this guide

- Use IPSec or SSL between the Web server and database server to protect the clear text credentials passed over the network.

Flowing the original caller to the database

In this scenario, calls are made from the Web application to the database using the security context of the original caller. With this approach, it's important to note the following:

- If you choose this approach, you need to use either Kerberos authentication (with accounts configured for delegation) or Basic authentication.

  A delegation scenario is discussed in the "Flowing the Original Caller to the Database" section later in this chapter.

- You must also enable impersonation in ASP.NET. This means that local system resource access is performed using the original caller's security context and as a result, ACLs on local resources such as the registry and event log require appropriate configuration.
- Database connection pooling is limited because original callers won't be able to share connections. Each connection is associated with the caller's security context.
- An alternate approach to flowing the user's security context is to flow the original caller's identity at the application level (for example, by using method and stored procedure parameters).

ASP.NET to Enterprise Services to SQL Server

In this scenario, ASP.NET pages call business components hosted in an Enterprise Services application that in turn connects to a database. As an example, consider an internal purchase order system that uses transactions over the intranet and allows internal departments to place orders. This scenario is shown in Figure 5.3.

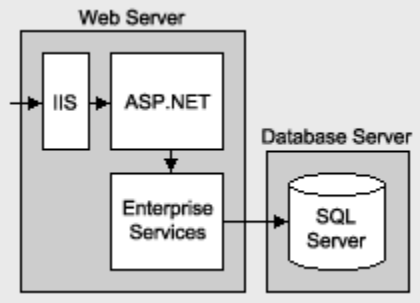

Figure 5.3. ASP.NET calls a component within Enterprise Services, which calls the database.

Characteristics

This scenario has the following characteristics:

- Users have Internet Explorer.
- Components are deployed on the Web server.
- The application handles sensitive data that must be secured while in transit.
- Business components connect to SQL Server using Windows authentication.
- Business functionality within these components is restricted based on the identity of the caller.

- Serviced components are configured as a server application (out-of-process).
- Components connect to the database using the server application's process identity.
- Impersonation is enabled within ASP.NET (to facilitate Enterprise Services role-based security).

Secure the Scenario

In this scenario, the Web server authenticates the original caller and flows the caller's security context to the serviced component. The serviced component authorizes access to business functionality based on the original caller's identity. The database authenticates against the Enterprise Service application's process identity (that is,. the database trusts the serviced components within the Enterprise Services application). When the serviced component makes calls to the database, it passes the user's identity at the application level (by using trusted query parameters).

Table 5.2. Security measures

Category	Detail
Authentication	Provide strong authentication at the Web server using Integrated Windows authentication. Flow the original caller's security context to the serviced component to support Enterprise Services (COM+) role checks. Secure connections to the database use Windows authentication. The database trusts the serviced component's identity to make the database calls. The database authenticates the Enterprise Services application process identity.
Authorization	Authorize access to business logic using Enterprise Services (COM+) roles.
Secure Communication	Secure sensitive data sent between the users and the Web application by using SSL. Secure sensitive data sent between the Web server and the database by using IPSec.

The Result

Figure 5.4 shows the recommended security configuration for this scenario.

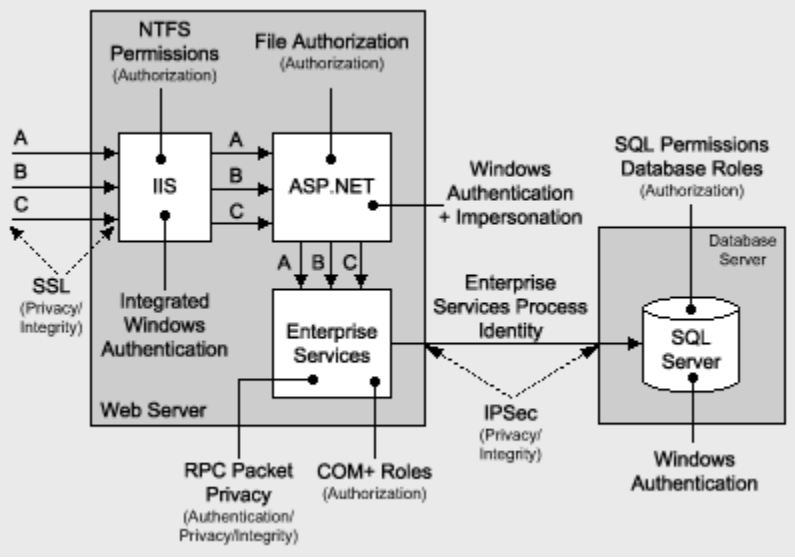

Figure 5.4. The recommended security configuration for the ASP.NET to local Enterprise Services to SQL Server intranet scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Creating a least privileged database account (see Chapter 12, Data Access Security)
- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)
- Configuring Enterprise Services security (see How To: Use Role-based Security with Enterprise Services in the Reference section of this guide)

Configuring IIS

Step	More Information
Disable Anonymous access for your Web application's virtual root directory Enable Integrated Windows Authentication	

Configuring ASP.NET

Step	More Information
Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your application's virtual directory root. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure your ASP.NET Web application for impersonation	Edit Web.config in your Web application's virtual directory Set the <**identity**> element to: `<identity impersonate="true" />`

Configure ASP.NET DCOM security to ensure that calls to Enterprise Services support caller impersonation	Edit Machine.config and locate the <**processModel**> element. Confirm that the **comImpersonationLevel** attribute is set to **Impersonate** (this is the default setting)
	
	`<processModel`
	
	`comImpersonationLevel="Impersonate"`

Configuring enterprise services

Step	More Information
Create a custom account for running Enterprise Services	**Note:** If you use a local account, you must also create a duplicate account on the SQL Server computer.
Configure the Enterprise Services application as a server application	This can be configured using the Component Services tool, or via the following .NET attribute placed in the service component assembly. `[assembly:`

	`ApplicationActivation(ActivationOption.`
	
	`Server)]`

Configure Enterprise Services (COM+) roles	Use the Component Services tool or script to add Windows users and/or groups to roles. Roles can be defined using .NET attributes within the serviced component assembly.
Configure Enterprise Services to run as your custom account	This must be configured using the Component Services tool or script. You cannot use .NET attributes within the serviced component assembly.

Configuring SQL Server

Step	More Information
Create a Windows account on your SQL Server computer that matches the Enterprise Services process account	The user name and password must match your custom Enterprise Services account. Give the account the following privileges: • Access this computer from the network • Deny logon locally • Log on as a batch job
Configure SQL Server for Windows authentication	
Create a SQL Server Login for your Enterprise Services account	This grants access to the SQL Server.
Create a new database user and map the login name to the database user	This grants access to the specified database.
Create a new database user role and add the database user to the role	
Establish database permissions for the database user role	Grant minimum permissions For details, see Chapter 12, Data Access Security

Configuring secure communication

Step	More Information
Configure the Web site for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure IPSec between Web server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Analysis

- ASP.NET and Enterprise Services are running as least privileged accounts, so potential damage from compromise is mitigated. If either process identity were compromised, the account's limited privileges reduce the scope of damage. Also, in the case of ASP.NET, if malicious script were injected, potential damage is constrained.
- The ASP.NET application must be configured for impersonation in order to flow the security context of the original caller to the Enterprise Services components (to support Enterprise Services (COM+) role-based

authorization). If you do not impersonate, role checks are made against the process identity (that is, the ASP.NET worker process). Impersonation affects who you authorize resources against.

- Without impersonation, system resource checks are against the ASP.NET process identity. With impersonation, system resource checks are made against the original caller. For more information about accessing system resources from ASP.NET, see Accessing System Resources in Chapter 8, "ASP.NET Security."

- By using Enterprise Services (COM+) roles, access checks are pushed to the middle tier, where the business logic is located. In this case, callers are checked at the gate, mapped to roles, and calls to business logic are based on roles. This avoids unnecessary calls to the back end. Another advantage of Enterprise Services (COM+) roles is that you can create and administer roles at deployment rime, using the Component Services Manager.

- Windows authentication to SQL means you avoid storing credentials in files and sending them across the network.

- The use of a local account to run the Enterprise Services application, together with a duplicated account on the database server, also works in the presence of a firewall where the ports required for Windows authentication may not be open. The use of Windows authentication and domain accounts may not work in this scenario.

Pitfalls

- The use of a duplicated Windows account on the database server (one that matches the Enterprise Services process account) results in increased administration. Passwords should be manually updated and synchronized on a periodic basis.

- Because .NET role-based security is based on Windows group membership, this solution relies on Windows groups being set up at the correct level of granularity to match the categories of users (sharing the same security privileges) who access the application.

ASP.NET to Web Services to SQL Server

In this scenario, a Web server that runs ASP.NET pages connects to a Web service on a remote server. This server in turn connects to a remote database server. As an example, consider a HR Web application that provides sensitive data specific to a user. The application relies on the Web service for data retrieval. The basic model for this application scenario is shown in Figure 5.5.

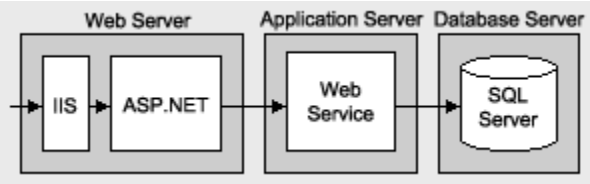

Figure 5.5. ASP.NET to remote Web Service to SQL Server

The Web service exposes a method that allows an individual employee to retrieve his or her own personal details. Details must be provided only to authenticated individuals using the Web application. The Web service also provides a method that supports the retrieval of any employee details. This functionality must be available only to members of the HR or payroll department. In this scenario, employees are categorized into three Windows groups:

- **HRDept** (members of the HR department)

  Members of this group can retrieve details about any employee.

- **PayrollDept** (members of the Payroll department)

  Members of this group can retrieve details about any employee.

- **Employees** (all employees)

  Members of this group can only retrieve their own details.

Due to the sensitive nature of the data, the traffic between all nodes should be secure.

Characteristics

- Users have Internet Explorer 5.x or later.
- All computers run Windows 2000 or later.
- User accounts are in Active Directory within a single forest.
- The application flows the original caller's security context all the way to the database.
- All tiers use Windows authentication.
- Domain user accounts are configured for delegation.
- The database does not support delegation.

Secure the Scenario

In this scenario, the Web server that hosts the ASP.NET Web application authenticates the original caller's identity and flows their security context to the remote server that hosts the Web service. This enables authorization checks to be applied to Web methods to either allow or deny access to the original caller. The database authenticates against the Web service process identity (the database trusts the Web service). The Web service in turn makes calls to the database and passes the user's identity at the application level using stored procedure parameters.

Table 5.3. Security measures

Category	Detail
Authentication	The Web application authenticates users by using Integrated Windows authentication from IIS. The Web service uses Integrated Windows authentication from IIS. It authenticates the original caller's security context delegated by the Web application. The Kerberos authentication protocol is used to flow the original caller security context from the Web application to the Web service using delegation. Windows authentication is used to connect to the database using the ASP.NET process account.
Authorization	The Web application performs role checks against the original caller to restrict access to pages. Access to the Web service methods is controlled by using .NET roles based on the original caller's Windows group membership.
Secure Communication	Sensitive data sent between the original callers and the Web application and Web service is secured by using SSL. Sensitive data sent between the Web service and the database is secure by using IPSec.

The Result

Figure 5.6 shows the recommended security configuration for this scenario.

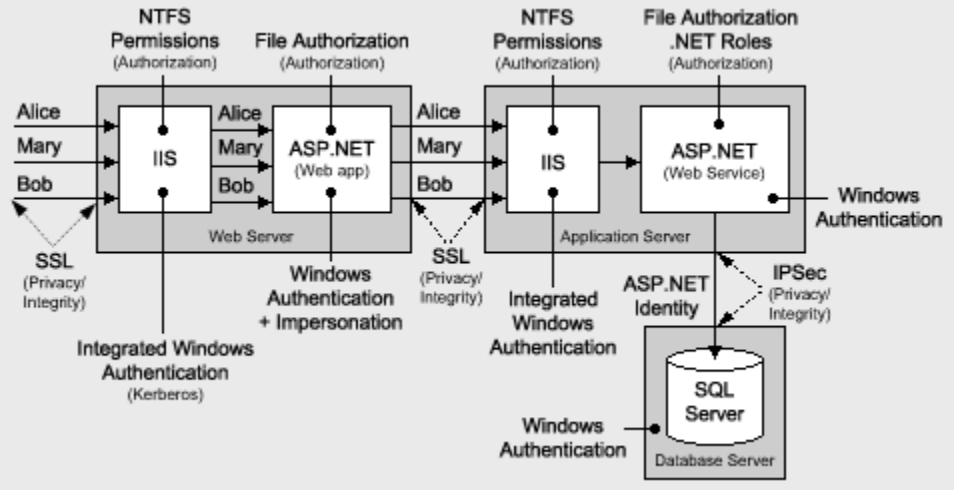

Figure 5.6. The recommended security configuration for the ASP.NET to Web Service to SQL Server intranet scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)

Configuring the Web server (that hosts the Web application)

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web application's virtual root directory Enable Windows Integrated Authentication for your Web application's virtual root	
Configure ASP.NET	
Step	**More Information**
Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your Web application's virtual directory Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure your ASP.NET Web application for impersonation	Edit Web.config in your Web application's virtual directory Set the <**identity**> element to: `<identity impersonate="true" />`

Configuring the application server (that hosts the Web service)

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web service's virtual root directory Enable Windows Integrated Authentication for your Web service's virtual root directory	
Configure ASP.NET	
Step	**More Information**

Change the ASPNET password to a known value	ASPNET is a least privileged local account used by default to run the ASP.NET Web applications.
	Set the ASPNET account's password to a know value by using Local Users and Groups.
	Edit Machine.config located in %windir%\Microsoft.NET\Framework\ v1.0.3705\CONFIG
	and reconfigure the password attribute on the <**processModel**> element: Default
	
	`<!-- userName="machine" password="AutoGenerate" -- >`
	
	Becomes
	
	`<!-- userName="machine" password="YourNewStrongPassword" -- >`

Configure your ASP.NET Web service to use Windows authentication	Edit Web.config in your Web service's virtual directory Set the <**authentication**> element to: `<authentication mode="Windows" />`
Make sure impersonation is off	Impersonation is off by default; however, double check to ensure that it's turned off in Web.config, as follows: `<identity impersonate="false" />` Note that because impersonation is disabled by default, the same effect can be achieved by removing the <**identity**> element.

Configure SQL Server

Step	**More Information**
Create a Windows account on your SQL Server computer that matches the ASP.NET process account used to run the Web service	The user name and password must match your custom ASP.NET account. Give the account the following privileges: • Access this computer from the network

	• Deny logon locally • Log on as a batch job
Configure SQL Server for Windows authentication	
Create a SQL Server Login for your custom ASP.NET account	This grants access to the SQL Server.
Create a new database user and map the login name to the database user	This grants access to the specified database.
Create a new database user role and add the database user to the role	
Establish database permissions for the database user role	Grant minimum permissions

Configuring secure communication

Step	More Information
Configure the Web site on the Web server for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure IPSec between Web server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers"in the Reference section of this guide.

Analysis

- Integrated Windows authentication in IIS is ideal in this scenario because all users are using Windows 2000 or later, Internet Explorer 5.x or later, and have accounts in Active Directory, which makes it possible to use the Kerberos authentication protocol (which supports delegation). This allows you to flow the security context of the user across computer boundaries.
- End user accounts must be NOT marked as "Sensitive and cannot be delegated" in Active Directory. The Web server computer account must be marked as "Trusted for delegation" in Active Directory. For more details, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.
- ASP.NET on the Web server and application server runs with a least privileged local account (the local ASPNET account), so potential damage from compromise is mitigated.
- The Web service and Web application are both configured for Windows authentication. IIS on both computers is configured for Integrated Windows authentication.
- When making a call to the Web service from the Web application, no credentials are passed by default. They are required in order to respond to the network authentication challenge issued by IIS on the downstream Web server. You must specify this explicitly by setting the **Credentials** property of the Web service proxy as shown in the following:

```
wsproxy.Credentials = CredentialCache.DefaultCredentials;
```

For more information about calling Web services with credentials, see Chapter 10, Web Services Security.

- The Web application is configured for impersonation. As a result, calls from the Web application to the Web service flow the original caller's security context and allow the Web service to authenticate (and authorize) the original caller.
- .NET roles are used within the Web service to authorize the users based on the Windows group to which they belong (HRDept, PayrollDept and Employees). Members of HRDept and PayrollDept can retrieve employee details for any employee, while members of the Employees group are authorized to retrieve only their own details.

Web methods can be annotated with the **PrincipalPermissionAttribute** class to demand specific role membership, as shown in the following code sample. Notice that **PrincipalPermission** can be used instead of **PrincipalPermissionAttribute**. This is a common feature of all .NET attribute types.

```
 [WebMethod]

[PrincipalPermission(SecurityAction.Demand,

                  Role=@"DomainName\HRDept)]

public DataSet RetrieveEmployeeDetails()

{

}
```

The attribute shown in the preceding code means that only members of the DomainName\HRDept Windows group are allowed to call the **RetrieveEmployeeDetails** method. If any nonmember attempts to call the method, a security exception is thrown.

- ASP.NET File Authorization (within the Web application and Web service) performs ACL checks against the caller for any file type for which a mapping exists in the IIS Metabase that maps the file type to Aspnet_isapi.dll. Static file types (such as .jpg, .gif, .htm, and so on), for which an ISAPI mapping does not exist are checked by IIS (again using the ACL attached to the file).
- Because the Web application is configured for impersonation, resources accessed by the application itself must be configured with an ACL that grants at least read access to the original caller.
- The Web service does not impersonate or delegate; therefore, it accesses local system resources and the database using the ASP.NET process identity. As a result, all calls are made using the single process account. This enables database connection pooling to be used. If the database doesn't support delegations (such as SQL Server 7.0 or earlier), this scenario is a good option.
- Windows authentication to SQL Server means you avoid storing credentials on the Web server and it also means that credentials are not sent across the network to the SQL Server computer.
- SSL between the original caller and Web server protects the data passed to and from the Web application.
- IPSec between the downstream Web server and database protects the data passed to and from the database.

Pitfalls

- The use of a duplicated Windows account on the database server (one that matches the ASP.NET process account) results in increased administration. Passwords should be manually updated and synchronized on a periodic basis.

  As an alternative, consider using least-privileged domain accounts. For more information about choosing an ASP.NET process identity, see Chapter 9, ASP.NET Security.

- Because .NET role-based security is based on Windows group membership, this solution relies on Windows groups being set up at the correct level of granularity to match the categories of users (sharing the same security privileges) who will access the application.
- Kerberos delegation is unrestricted and as a result you must carefully control which applications identities run on the Web server. To raise the bar on security, limit the scope of the domain account's reach by removing the account from Domain Users group and provide access only from appropriate computers. For more information, see Default Access Control Settings white paper.

Q&A

- **The database doesn't know who the original caller is. How can I create an audit trail?**

Audit end user activity within the Web service or pass the identity of the user explicitly as a parameter of the data access call.

Related scenarios

If you need to connect to non-SQL Server databases, or you currently use SQL authentication, you must pass database account credentials explicitly using the connection string. If you do so, make sure that you securely store the connection string.

For more information, see Storing Database Connection Strings Securely within Chapter 12, "Data Access Security."

ASP.NET to Remoting to SQL Server

In this scenario, a Web server that runs ASP.NET pages makes secure connections to a remote component on a remote application server. The Web server communicates with the component by using .NET Remoting over the HTTP channel. The remote component is hosted by ASP.NET. This is shown in Figure 5.7.

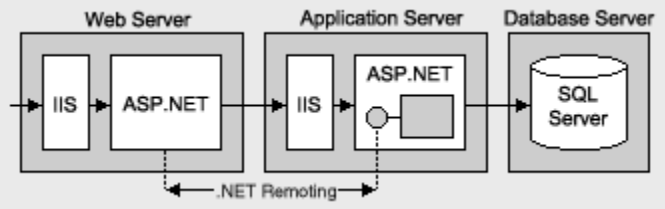

Figure 5.7. ASP.NET to remoting using .NET Remoting to SQL Server

Characteristics

- Users have various types of Web browser.
- The remote component is hosted by ASP.NET.
- The Web application communicates with the remote component using the HTTP channel.
- The ASP.NET application calls the .NET remote component and passes the original caller's credentials for authentication. These are available from Basic authentication.
- The data is sensitive and therefore must be secured between processes and computers.

Secure the Scenario

In this scenario, the Web server that hosts the ASP.NET Web application authenticates the original callers. The Web application is able to retrieve the caller's authentication credentials (user name and password) from HTTP server variables. It can then use them to connect to the application server that hosts the remote component, by configuring the remote component proxy. The database uses Windows authentication to authenticate against the ASP.NET process identity (that is, the database trusts the remote component). The remote component in turn calls the database and passes the original caller's identity at the application level using stored procedure parameters.

Table 5.4. Security measures

Category	Detail
Authentication	Authenticate users using Basic authentication from IIS (in addition to SSL). Use Windows authentication from remote component (ASP.NET/IIS). Use Windows authentication to connect to the database using a least

	privileged ASP.NET account.
Authorization	ACL checks against original caller on the Web server. Role checks within the remote component against original caller. Database permissions against the ASP.NET (remote component) identity.
Secure Communication	Secure sensitive data sent between the users and the Web application and remote objects hosted in IIS using SSL. Secure sensitive data sent between the Web server and the database using IPSec.

The Result

Figure 5.8 shows the recommended security configuration for this scenario.

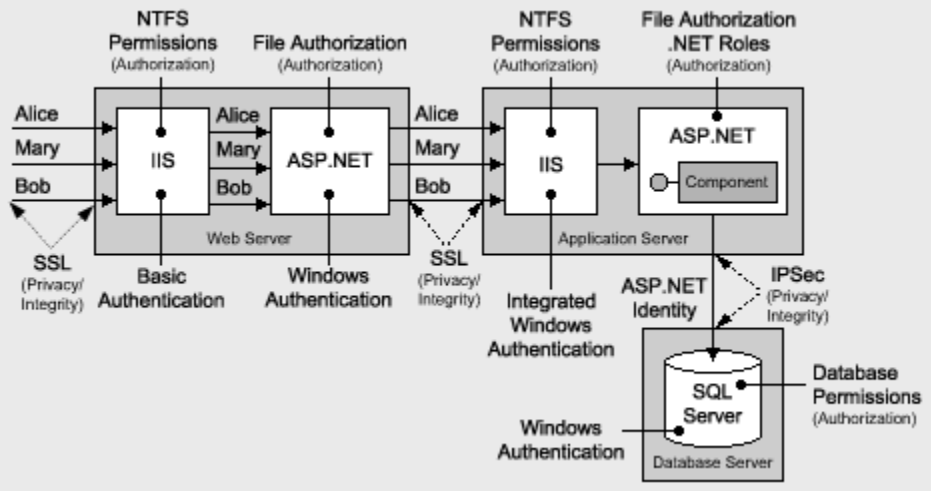

Figure 5.8. The recommended security configuration for the ASP.NET to remote Web Service to SQL Server intranet scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Creating a least privileged database account (see Chapter 12, Data Access Security)
- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)

Configuring the Web server

Configure IIS	
Step	More Information
Disable Anonymous access for your Web application's virtual root directory	

Enable Basic authentication	Use SSL to protect the Basic authentication credentials.
Configure ASP.NET	
Step	**More Information**
Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your application's virtual directory root Set the <**authentication**> element to: `<authentication mode="Windows" />`

Configure the application server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web application's virtual root directory Enable Integrated Windows authentication	
Configure ASP.NET	
Step	**More Information**
Configure your remote component (within ASP.NET) to use Windows authentication	Edit Web.config in your remote component's virtual directory root Set the <**authentication**> element to: `<authentication mode="Windows" />`

Change the ASPNET password to a known value	ASPNET is a least privileged local account used by default to run ASP.NET Web applications (and in this case the remote component host process).
	Set the ASPNET account's password to a know value by using Local Users and Groups.
	Edit Machine.config located in %windir%\Microsoft.NET\Framework\v1.0.3705\CONFIG
	and reconfigure the password attribute on the <**processModel**> element Default
	`<!-- userName="machine" password="AutoGenerate" -- >`
	Becomes
	`<!-- userName="machine" password="YourNewStrongPassword" -- >`

Make sure impersonation is off	Impersonation is off by default; however, double check to ensure that its turned off in web.config, as shown below:

	`<identity impersonate="false" />` The same effect can be achieved by removing the <**identity**> element.

Configure SQL Server

Step	More Information
Create a Windows account on your SQL Server computer that matches the ASP.NET process account used to run the Web service	The user name and password must match your custom ASP.NET account. Give the account the following privileges: • Access this computer from the network • Deny logon locally • Log on as a batch job
Configure SQL Server for Windows authentication	
Create a SQL Server Login for your custom ASP.NET account	This grants access to the SQL Server
Create a new database user and map the login name to the database user	This grants access to the specified database
Create a new database user role and add the database user to the role	
Establish database permissions for the database user role	Grant minimum permissions

Configuring secure communication

Step	More Information
Configure the Web site on the Web server for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure the Web site on the application server for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure IPSec between application server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide."

Analysis

- ASP.NET on the Web server and application sever is running as a least privileged local account, so potential damage from compromise is mitigated. The default ASPNET account is used in both cases.

  Use of the ASPNET local account (duplicated on the SQL Server computer) further reduces the potential security risk. A duplicated Windows account on the database server allows the remote component to run with a least privilege ASP.NET account on the application server.

- Basic authentication at the Web server allows the user's credentials to be used by the Web application to respond to Windows authentication challenges from the application server.

  To call the remote component using the caller's credentials, the Web application configures the remote component proxy as shown in the following code fragment.

```
string pwd = Request.ServerVariables["AUTH_PASSWORD"];

string uid = Request.ServerVariables["AUTH_USER"];

IDictionary channelProperties =

                ChannelServices.GetChannelSinkProperties(proxy);

NetworkCredential credentials;

credentials = new NetworkCredential(uid, pwd);

ObjRef objectReference = RemotingServices.Marshal(proxy);

Uri objectUri = new Uri(objectReference.URI);

CredentialCache credCache = new CredentialCache();

credCache.Add(objectUri, "Negotiate", credentials);

channelProperties["credentials"] = credCache;

channelProperties["preauthenticate"] = true;
```

For more information about flowing security credentials to a remote component, see Chapter 11, .NET Remoting Security.

- Impersonation is not enabled within the ASP.NET Web application, because the remoting proxy is specifically configured using the user's credentials obtained by Basic authentication. Any other resource accessed by the Web application uses the security context provided by the ASP.NET process account.
- SSL between the user and Web server protects the data passed to and from the Web server and also protects the Basic credentials passed in clear text during the authentication process.
- Integrated Windows authentication at the application server provides .NET role checks against the original caller. The roles correspond to Windows groups.

  Role-based checks can be performed, even without impersonation.

- ASP.NET File Authorization performs ACL checks against the caller for any file type for which a mapping exists in the IIS Metabase that maps the file type to aspnet_isapi.dll. IIS performs access checks for static files (not mapped to an ISAPI extension within IIS).
- Because impersonation is not enabled on the application server, any local or remote resource access performed by the remote component does so using the ASPNET security context. ACLs should be set accordingly.
- Windows authentication to SQL Server means you avoid storing credentials on the application server and it also means that credentials are not sent across the network to the SQL Server computer.

Pitfalls

- The use of a duplicated Windows account on the database server (one that matches the ASP.NET process account) results in increased administration. Passwords should be manually updated and synchronized on a periodic basis.
- Because .NET role-based security is based on Windows group membership, this solution relies on Windows groups being set up at the correct level of granularity to match the categories of users (sharing the same security privileges) who will access the application.

Related scenarios

The Web server uses Kerberos to authenticate callers. Kerberos delegation is used to flow the original caller's security context across to the remote component on the application server.

This approach requires that all user accounts be configured for delegation. The Web application would also be configured for impersonation and would use DefaultCredentials to configure the remote component proxy. This technique is discussed further in the Flowing the Original Caller section of Chapter 11, ".NET Remoting Security."

Flowing the Original Caller to the Database

The scenarios discussed earlier have used the trusted subsystem model and in all cases the database has trusted the application server or Web server to correctly authenticate and authorize users. While the trusted subsystem model offers many advantages, some scenarios (perhaps for auditing reasons) may require you to use the impersonation/delegation model and flow the original caller's security context across computer boundaries all the way to the database.

Typical reasons why you may need to flow the original caller to the database include:

- You need granular access in the database and permissions are restricted by object. Specific users or groups can read, while others can write to individual objects.

  This is in contrast to less granular task-based authorization, where role membership determines read and write capabilities for specific objects.

- You may want to use the auditing capabilities of the platform, rather than flow identity and perform auditing at the application level.

If you do choose an impersonation/delegation model (or are required to do so due to corporate security policy) and flow the original caller's context through the tiers of your application to the back end, you must design with delegation and network access in mind (which is nontrivial when spanning multiple computers). The pooling of shared resources (such as database connections) also becomes a key issue and can significantly reduce application scalability.

This section shows you how to implement the impersonation/delegation for two of the most common application scenarios:

- ASP.NET to SQL Server
- ASP.NET to Enterprise Services to SQL Server

For more information about the trusted subsystem and impersonation/delegation models and their relative merits, see Chapter 3, Authentication and Authorization.

ASP.NET to SQL Server

In this scenario, calls to the database are made using the security context of the original caller. Authentication options described in this section include Basic and Integrated Windows authentication. A Kerberos delegation scenario is described within the "ASP.NET to Enterprise Services to SQL Server" section.

Using basic authentication at the Web server

The following configuration settings for Basic authentication enable you to flow the original caller all the way to the database.

Table 5.5. Security measures

Category	Detail
Authentication	Authenticate users by using Basic authentication from IIS. Use Windows authentication within ASP.NET.

	Turn on impersonation in ASP.NET. Use Windows authentication to communicate with SQL Server.
Authorization	Use ACL checks against the original caller on the Web server. If the original callers are mapped to Windows groups (based on application requirements, for example, Managers, Tellers, and so on) then you can use .NET role checks against the original caller to restrict access to methods.
Secure Communication	Secure the clear text credentials sent between the Web server and the database by using SSL. To secure all sensitive data sent between the Web application and database, use IPSec.

With this approach, it's important to note the following points:

- Basic authentication prompts the user with a pop-up dialog box into which they can type credentials (user name and password).
- The database must recognize the original caller. If the Web server and database are in different domains, appropriate trust relationships must be enabled to allow it to authenticate the original caller.

Using integrated Windows authentication at the Web server

Integrated Windows authentication results in either NTLM or Kerberos authentication and is dependent upon the client and server computer configurations.

NTLM authentication does not support delegation and as a result does not allow you to flow the original caller's security context from the Web server to a physically remote database. The single network hop allowed for NTLM authentication is used between the browser and Web server. To use NTLM authentication, the SQL Server must be installed on the Web server, which is likely to be appropriate only for very small intranet applications.

ASP.NET to Enterprise Services to SQL Server

- In this scenario, ASP.NET pages call business components hosted in a remote Enterprise Services application that in turn talk to a database. The original caller's security context flows all the way from the browser to the database. This is shown in Figure 5.9.

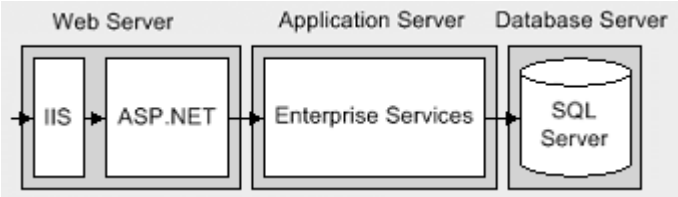

Figure 5.9. ASP.NET calls a component within Enterprise Services, which calls the database

Characteristics

- Users have Internet Explorer 5.x or later.
- All computers are Windows 2000 or later.
- User accounts are maintained in Active Directory within a single forest.
- The application flows the original caller's security context (at the operating system level) all the way to the database.

- All tiers use Windows authentication.
- Domain user accounts are configured for delegation and the account used to run the Enterprise Services application must be marked as "Trusted for delegation" within Active Directory.

Secure the scenario

In this scenario, the Web server authenticates the caller. You must then configure ASP.NET for impersonation in order to flow the original caller's security context to the remote Enterprise Services application. Within the Enterprise Services application, component code must explicitly impersonate the caller (using **CoImpersonateClient**) in order to ensure the caller's context flows to the database.

Table 5.6. Security measures

Category	Detail
Authentication	All tiers support Kerberos authentication (the Web server, the application server, and database server).
Authorization	Authorization checks are performed in the middle tier with Enterprise Services (COM+) roles against the original caller's identity.
Secure Communication	SSL is used between the browser and the Web server to secure sensitive data. RPC Packet Privacy (providing encryption) is used between ASP.NET and the serviced components within the remote Enterprise Services application. IPSec is used between the serviced components and the database.

The Result

Figure 5.10 shows the recommended security configuration for this scenario.

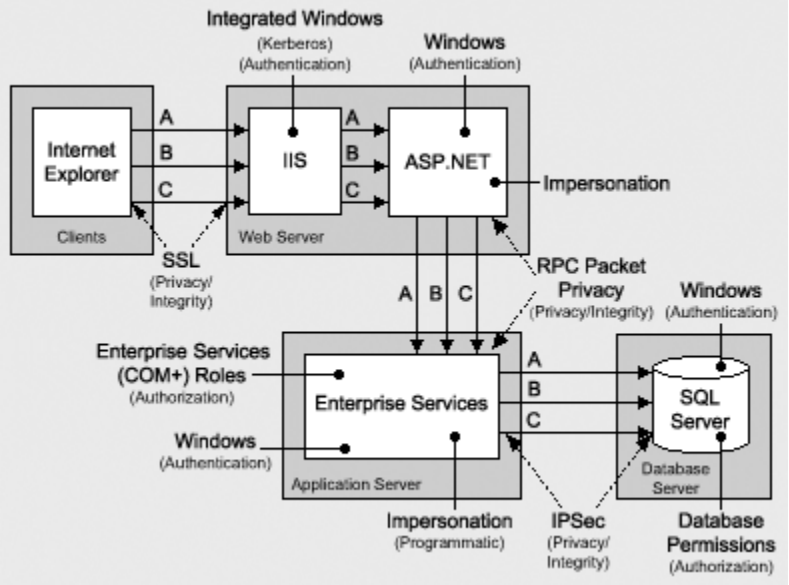

Figure 5.10. ASP.NET calls a component within Enterprise Services, which calls the database. The original caller's security context flows to the database.

Security configuration steps

Before you begin, you should be aware of the following configuration issues:

- The Enterprise Services process account must be marked "Trusted for delegation" in Active Directory and end user accounts must not be marked "Sensitive and cannot be delegated." For more information, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.
- Windows 2000 or later is required on all computers. This includes client (browser) computers and all servers.
- All computers must be in the Active Directory and must be part of a single forest.
- The application server that hosts Enterprise Services must be running Windows 2000 SP3.
- If you are using Internet Explorer 6.0 on Windows 2000, it defaults to NTLM authentication instead of the required Kerberos authentication. To enable Kerberos delegation, see article Q299838, Can't Negotiate Kerberos Authentication After Upgrading to Internet Explorer 6, in the Microsoft Knowledge Base.

Configure the Web Server (IIS)	
Step	**More Information**
Disable Anonymous access for your Web application's virtual root directory Enable Windows Integrated authentication	
Configure the Web Server (ASP.NET)	
Step	**More Information**
Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your Web application's virtual directory root Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure your ASP.NET Web application for impersonation	Edit Web.config in your Web application's virtual directory Set the <**identity**> element to: `<identity impersonate="true" />`
Configure the DCOM impersonation level used by the ASP.NET Web application for outgoing calls	The ASP.NET Web application calls the remote serviced components over DCOM. The default impersonation level used for outgoing calls from ASP.NET is Impersonate. This must be changed to Delegate in Machine.config. Edit Machine.config, locate the <**processModel**> element, and set the **comImpersonateLevel** attribute to "Delegate" as shown below. `<processModel comImpersonationLevel="Delegate"`

Configure the DCOM authentication level at the client	DCOM authentication levels are determined by both client and server. The DCOM client in this case is ASP.NET.
	
	Edit Machine.config, locate the <**processModel**> element and set the **comAuthenitcationLevel** attribute to "**PktPrivacy**" as shown below.
	
	`<processModel`
	
	`comAuthenticationLevel="PktPrivacy"`

Configure Serviced Components (and the Application Server)	
Step	**More Information**
Managed class(es) must inherit from the ServicedComponent class	See article Q306296, HOW TO: Create a Serviced .NET Component in Visual C# .NET, in the Microsoft Knowledge Base.

Add code to the serviced component to impersonate the caller by calling the **CoImpersonateClient()** and **CoRevertToSelf()** APIs from OLE32.DLL before accessing remote resources (for example, a database) in order for the caller's context to be used. By default, the Enterprise Services process context is used for outgoing calls.	Add references to OLE32.DLL:
	`class COMSec`
	`{`
	`[DllImport("OLE32.DLL", CharSet=CharSet.Auto)]`
	`public static extern long CoImpersonateClient();`
	`[DllImport("OLE32.DLL", CharSet=CharSet.Auto)]`
	`public static extern long CoRevertToSelf();`
	`}`
	Call these external functions before calling remote resources:
	`COMSec.CoImpersonateClient();`
	`COMSec.CoRevertToSelf();`
	For more information, see Chapter 9, Enterprise Services Security.

Configure the Enterprise Services application as a server application	This can be configured using the Component Services tool, or via the following .NET attribute placed in the service component assembly.
	`[assembly:`
	`ApplicationActivation`
	`(ActivationOption.Server)]`

Configure the Enterprise Services application to use packet privacy authentication (to provide secure communication with encryption)	Add the following .NET attribute to the serviced component assembly.
	`[assembly: ApplicationAccessControl(`
	`Authentication =`
	`AuthenticationOption.Privacy)]`

Configure the Enterprise Services application for component level role-based security	To configure role checks at the process and component level (including interfaces and classes) use the following attribute.

	`[assembly:`
	`ApplicationAccessControl(AccessChecksLevel=`
	`AccessChecksLevelOption. ApplicationComponent)]`
	
	Decorate classes with the following attribute:
	
	`[ComponentAccessControl(true)]`
	
	For more information about configuring interface and method level role checks, see Configuring Security in Chapter 9, "Enterprise Services Security."

Create a custom account for running Enterprise Services and mark it as Trusted for delegation in Active Directory	The Enterprise Services application needs to run as domain account marked as Trusted for Delegation in Active Directory. For more information, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.
Configure Enterprise Services to run as your custom account	This must be configured using the Component Services tool or script. You cannot use .NET attributes within the serviced component assembly.
Configure the Database Server (SQL Server)	
Step	**More Information**
Configure SQL Server for Windows authentication	
Create SQL Server Logins for the Windows groups that the users belong to.	This grants access to the SQL Server. The access control policy treats Windows groups as roles. For example, you may have groups such as **Employees**, **HRDept** and **PayrollDept**.
Create new database users for each SQL Server login	This grants access to the specified database.
Establish database permissions for the database users	Grant minimum permissions For more information, see Chapter 12, Data Access Security.

Analysis

- The key to flowing the original caller's security context is Kerberos authentication, which generates a delegate-level token. After the server process (IIS) receives the delegate-level token, it can pass it to any other process, running under any account on the same computer, without changing its delegation level. It does not matter whether the worker process is running as a local or domain account. It *does* matter what IIS is running as. If it's running as something other than **LocalSystem**, the account it is running under needs to be marked as "Trusted for delegation" in Active Directory.

  If IIS is running as **LocalSystem**, the computer account must be marked as "Trusted for delegation". For more information, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.

- Integrated Windows authentication (with Kerberos) in IIS is ideal in this scenario because all users have Windows accounts and they are using Internet Explorer 5.x or later. The benefit of Integrated Windows authentication is that the user's password is never sent over the wire. Additionally, the logon will be transparent because Windows will use the current interactive user's logon session.

- ASP.NET constructs a **WindowsPrincipal** object and attaches it to the current Web request context (**HttpContext.User**). If you need to perform authorization checks within the Web application you can use the following code.

```
WindowsPrincipal wp = (HttpContext.Current.User as
  WindowsPrincipal);
if ( wp.IsInRole("Manager") )
{
  // User is authorized to perform manager-specific functionality
}
```

The ASP.NET **FileAuthorizationModule** provides ACL checks against the original caller for ASP.NET file types that are mapped within IIS to the Aspnet_isapi.dll. For static file types such as .jpg, .gif and .htm files, IIS acts as the gatekeeper and performs access checks using the original caller's identity.

- By using Windows authentication to SQL, you avoid storing credentials in files on the application server and avoid passing them across the network. For example include the Trusted_Connection attribute in the connection string:

```
ConStr="server=YourServer; database=yourdatabase;
Trusted_Connection=Yes;"
```

- The original caller's context flows across all tiers, which makes auditing extremely easy. You can use platform-level auditing (for example, auditing features provided by Windows and SQL Server).

Pitfalls

- If you are using Internet Explorer 6.0 on Windows 2000, the default authentication mechanism that is negotiated is NTLM (and not Kerberos). For more information, see article Q299838, Can't Negotiate Kerberos Authentication After Upgrading to Internet Explorer 6, in the Microsoft Knowledge Base.
- Delegating users across tiers is expensive in terms of performance and application scalability compared to using the trusted subsystem model. You cannot take advantage of connection pooling to the database, because connections to the database are tied to original caller's security context and therefore cannot be efficiently pooled.
- This approach also relies on the granularity of Windows groups matching your application's security needs. That is, Windows groups must be set up at the correct level of granularity to match the categories of users (sharing the same security privileges) who access the application.

Summary

This chapter has described how to secure a set of common intranet application scenarios.

For Extranet and Internet application scenarios, see Chapter 6, Extranet Security and Chapter 7, Internet Security.

Extranet Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter describes how to secure common extranet application scenarios. It presents the characteristics of each scenario and describes the steps necessary to secure the scenario. Analysis sections are also included to provide further information. (14 printed pages)

Contents

Extranet applications are those that share resources or applications across two different companies or divisions. The applications and resources are exposed over the Internet. One of the main challenges associated with extranet applications is developing an authentication approach that both parties agree to. Your choices may be limited in this respect because you may need to interoperate with existing authentication mechanisms.

Extranet applications generally share some common characteristics:

- You have tighter control over user accounts, compared to Internet scenarios.
- You may have a higher level of trust for the user accounts, compared to applications that have Internet users.

The scenarios presented in this chapter that are used to illustrate recommended authentication, authorization, and secure communication techniques include:

- Exposing a Web Service
- Exposing a Web Application

Exposing a Web Service

Consider a business-to-business partner exchange scenario where a publisher company publishes and sells its services over the Internet. It exposes information to selected partner companies using a Web service. Users within each partner company access the Web service using an Intranet-based internal Web application. This scenario is shown in Figure 6.1.

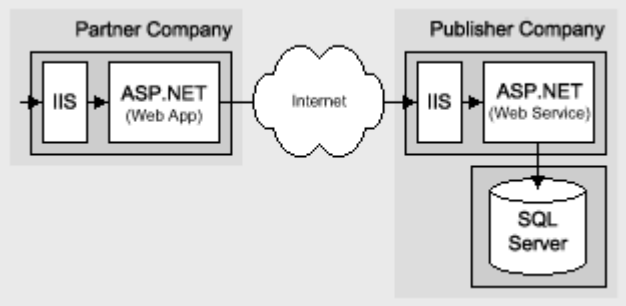

Figure 6.1. Extranet Web service business-to-business partner exchange

Characteristics

This scenario has the following characteristics:

- The publisher company exposes a Web service over the Internet.
- Partner company (not individual user) credentials (X.509 client certificates) are validated by the publisher to authorize access to resources. The publisher does not need to know about the user's individual logins in the partner company.
- Client certificates are mapped to Active Directory® directory service accounts within the publisher company.
- The extranet contains a separate Active Directory from the (internal) corporate Active Directory. The extranet Active Directory is in a separate forest, which provides a separate trust boundary.
- Web service authorization is based on the mapped Active Directory account. You can use separate authorization decisions based on partner company identity (represented by separate Active Directory accounts per company).
- The database is accessed by a single trusted connection that corresponds to the ASP.NET Web service process identity.
- The data retrieved from the Web service is sensitive and must be secured while in transit (internally within the publisher company and externally while flowing over the Internet).

Secure the Scenario

In this scenario, each partner company's internal Web application retrieves data from the publisher company through the Web service and then presents the retrieved data to its users. The publisher requires a secure mechanism to authenticate partner companies, although the identity of individual users within partner companies is not relevant.

Due to the sensitive nature of the data sent between the two companies over the Internet, it must be secured using SSL while in transit.

A firewall that permits only inbound connections from the IP address of extranet partner companies is used to prevent other unauthorized Internet users from opening network connections to the Web service server.

Table 6.1. Security measures

Category	Detail
Authentication	Partner applications use client certificates with each request to the Web service. Client certificates from partner companies are mapped to individual Active Directory accounts. Microsoft® Windows® authentication is used at the database. The ASP.NET Web service process identity is used to connect. The database trusts the Web service.
Authorization	The Web service uses .NET role-based authorization to check that authenticated Active Directory accounts are members of a Partner group.
Secure Communication	SSL is used to secure the communication between the partner Web application and publisher's Web service. IPSec is used to secure all communication between the Web service and the database.

The Result

Figure 6.2 shows the recommended security configuration for this scenario.

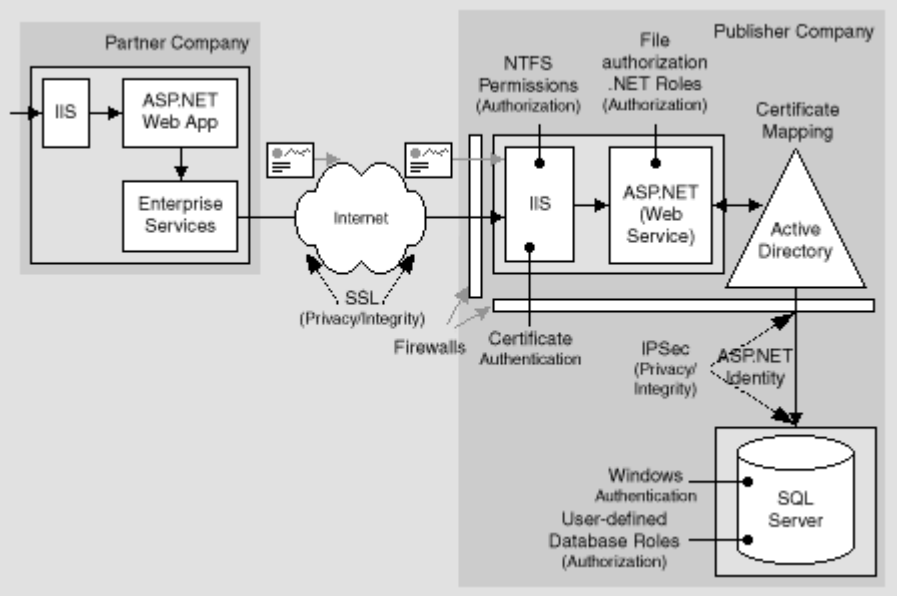

Figure 6.2. The recommended security configuration for the Web service business-to-business partner exchange scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Creating custom ASP.NET accounts (see How To: Create a Custom Account to Run ASP.NET in the Reference section of this guide)
- Creating a least privileged database account (see Chapter 12, Data Access Security)
- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)
- Configuring IPSec through firewalls (see article Q233256, How to Enable IPSec Traffic Through a Firewall, in the Microsoft Knowledge Base).
- Calling a Web service using SSL (see How To: Call a Web Service Using SSL in the Reference section of this guide); this solution technique is required within the partner company
- The discussion of certificate management and the infrastructure is beyond the scope of this topic, for more information search for Certificates and Authenticode on Microsoft TechNet.

Configuring the partner application

This chapter does not go into details about the partner application and its security configuration. However, the following points needs to be considered to facilitate communication between the partner application and Web service:

- The partner company's Web application can choose an authentication mechanism that allows it to authenticate and authorize its internal users. Those users are not passed to the Web service for further authentication.
- The partner company's Web application makes calls on behalf of its user to the Web service. Users cannot directly call the Web service.
- The partner company's Web application uses a client certificate to prove its identity to the Web service.

- If the partner application is an ASP.NET Web application, then it must use an intermediate out of process component (an Enterprise Services application or Windows service) to load the certificate and forward it to the Web service.

  For more information about why this is necessary and the steps to achieve this, see How to call a Web service using client certificates from ASP.NET in the Reference section of this guide.

Configuring the extranet Web server

Configure IIS	
Step	**More Information**
Disable Anonymous access for the Web service's virtual root directory. Enable certificate Authentication for your Web application's and Web service's virtual root.	To work with IIS authentication settings, use the IIS MMC snap-in. Select your application's virtual directory, right-click and then click **Properties**. Click the **Directory Security** tab, and then click **Edit** within the **Anonymous access and authentication control** group. See How To: Set Up SSL on a Web Server in the Reference section of this guide. See How To: Call a Web Service Using Client Certificates from ASP.NET in the Reference section of this guide.
Configure Active Directory (Extranet)	
Step	**More Information**
Set up Active Directory accounts to represent partner companies	A separate extranet Active Directory is used. This is located in its own forest, and is completely separate from the corporate Active Directory.
Configure certificate mapping	See the Step-by-Step Guide to Mapping Certificates to User Accounts on Microsoft TechNet. Also see article Q313070, HOW TO: Configure Client Certificate Mappings in IIS 5.0, in the Microsoft Knowledge Base.
Configure ASP.NET (Web service)	
Step	**More Information**
Configure the ASP.NET Web service to use Windows authentication	Edit Web.config in the Web service's virtual root directory Set the <**authentication**> element to: `<authentication mode="Windows" />`
Reset the password of the ASPNET account (used to run ASP.NET) to a known strong password	This allows you to create a duplicate local account (with the same username and password) on the database server. This is required to allow the ASPNET account to respond to network authentication challenges from the database server when it connects using Windows authentication. An alternative here is to use a least privileged domain account (if Windows authentication is permitted through the firewall). For more information, see Process Identity for ASP.NET in Chapter 8, "ASP.NET Security." Edit Machine.config located in %windir%\Microsoft.NET\Framework\v1.0.3705\CONFIG Set your custom account username and password attributes on the

	<**processModel**> element
	
	Default
	
	`<!-- userName="machine" password="AutoGenerate" -- >`
	
	Becomes
	
	`<!-- userName="machine" password="YourStrongPassword" -- >`

Configuring SQL Server

Step	More Information
Create a Windows account on the computer running Microsoft SQL Server™ that matches the ASP.NET process account used to run the Web service (by default ASPNET)	The user name and password must match your ASP.NET process account. Give the account the following privileges: • Access this computer from the network • Deny logon locally • Log on as a batch job
Configure SQL Server for Windows authentication	
Create a SQL Server Login for the ASPNET account	This grants access to the SQL Server.
Create a new database user and map the login name to the database user	This grants access to the specified database.
Create a new user-defined database role within the database and place the database user into the role	
Establish database permissions for the database role	Grant minimum permissions See Chapter 12, Data Access Security.

Configuring secure communication

Step	More Information
Configure the Web site on the Web server for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure IPSec between Web server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Analysis

- ASP.NET on the Web server is running as a least privileged local account (the default ASPNET account), so potential damage from compromise is mitigated.

- The ASP.NET Web applications within the partner companies use Windows Integrated authentication and perform authorization to determine who can access the Web service.
- The ASP.NET Web application within the partner company uses an intermediate Enterprise Services application to retrieve client certificates and make calls to the Web service.
- The publisher company uses the partner organization name (contained in the certificate) to perform certificate mapping within IIS.
- The Web service uses the mapped Active Directory account to perform authorization, using **PrincipalPermission** demands and .NET role checks.
- Windows authentication to SQL Server means you avoid storing credentials on the Web server and it also means that credentials are not sent across the internal network to the SQL Server computer. If you use SQL authentication, it is important to secure the connection string (containing a user name and password) within the application and as it is passed across the network. Use DPAPI or one of the alternative secure storage strategies discussed in Chapter 12, Data Access Security, to store connection strings and use IPSec to protect the connection string (and sensitive application data) as it is passed between the Web service and database.
- SSL between partner companies and Web service protects the data passed across the Internet.
- IPSec between the Web service and database protects the data passed to and from the database on the corporate network. In some scenarios where the partner and publisher communicate over a private network, it may be possible to use IPSec for machine authentication in addition to secure communication.

Pitfalls

- The use of a duplicated local Windows account on the database server (one that matches the ASP.NET process account local to IIS) results in increased administration. Passwords must be manually updated and synchronized on a periodic basis.
- Because .NET role-based security is based on Windows group membership, this solution relies on Windows groups being set up at the correct level of granularity to match the categories of users (sharing the same security privileges) who will access the application. In this scenario, Active Directory accounts must be a member of a Partner group.

Q&A

- **The database doesn't know who the original caller is. How can I create an audit trail?**

  Audit end user (partner company) activity within the Web service. Pass the partner company identity at the application level to the database using stored procedure parameters.

Related scenarios

The publisher company might publish non-sensitive data such as soft copies of magazines, newspapers, and so on. In this scenario, the publisher can provide a unique username and password for each partner to connect with to retrieve the data from the Web service.

In this related scenario, the publisher's Web site is configured to authenticate users with Basic authentication. The partner application uses the username and password to explicitly set the credentials for the Web service proxy.

More information

For more information about configuring Web service proxies, see Chapter 10, Web Services Security.

Exposing a Web Application

In this scenario the publisher company gives its partners exclusive access to its application over the Internet and provides a partner-portal application; for example, to sell services, keep partners updated with product information, and provide online collaboration and so on. This scenario is shown in Figure 6.3.

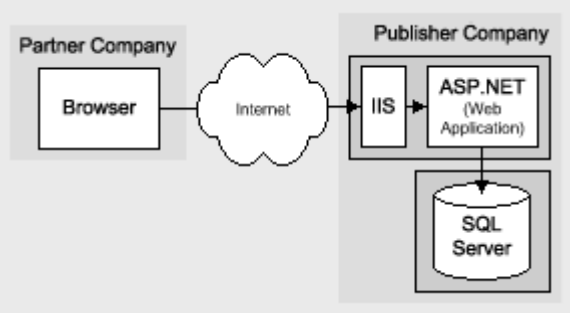

Figure 6.3. Partner portal scenario

Scenario Characteristics

This scenario has the following characteristics:

- The partner Web application accepts credentials either by using a Forms login page or it presents a login dialog using Basic authentication in IIS.
- The credentials are validated against a separate Active Directory within the extranet perimeter network (also known as DMZ, demilitarized zone, and screened subnet). The extranet Active Directory is in a separate forest, which provides a separate trust boundary.
- The database is accessed by a single trusted connection that corresponds to the ASP.NET Web application process identity.
- Web application authorization is based on either a **GenericPrincipal** object (created as part of the Forms authentication process) or a **WindowsPrincipal** object (if Basic authentication is used).
- The data retrieved from the Web application is sensitive and must be secured while in transit (internally within the publisher company and externally while flowing over the Internet).

Secure the Scenario

Due to the sensitive nature of the data sent between the two companies over the Internet, it must be secured using SSL while in transit.

- A firewall that permits only inbound connections from the IP address of extranet partner companies is used to prevent other unauthorized Internet users from opening network connections to the Web server.

Table 6.2. Security measures

Category	Detail
Authentication	Users within partner companies are authenticated by the Web application using either Basic or Forms authentication against the extranet Active Directory. Windows authentication is used at the database. The ASP.NET Web application process identity is used to connect. The database trusts the Web application.
Authorization	The Web application uses .NET role-based authorization to check that the authenticated user (represented by either a **GenericPrincipal** object or a **WindowsPrincipal** object, for Forms and Basic authentication respectively) is a member of a Partner group.
Secure Communication	SSL is used to secure the communication between the partner Web browser and publisher's Web application. IPSec is used to secure all communication between the Web application and the database.

The Result

Figure 6.4 shows the recommended security configuration for this scenario.

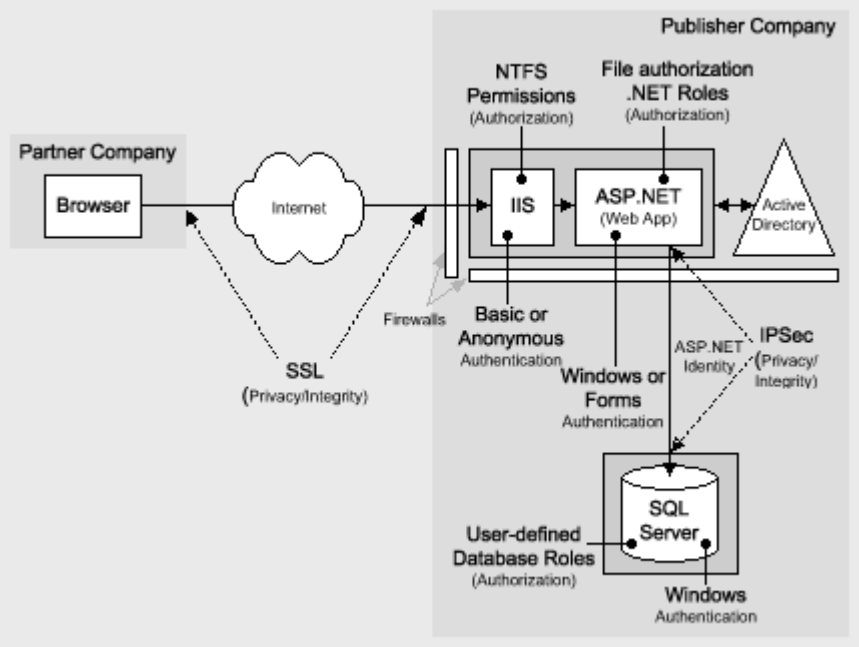

Figure 6.4. The recommended security configuration for the partner portal scenario

Configuring the extranet Web server

Configure IIS	
Step	**More Information**
To use Forms authentication, enable Anonymous access for the Web application's virtual root directory —or— To use Basic authentication, disable Anonymous access and select Basic authentication	
Configure Active Directory (Extranet)	
Step	**More Information**
Set up Active Directory accounts to represent partner users	A separate extranet Active Directory is used. This is located in its own forest and is completely separate from the corporate Active Directory.
Configure ASP.NET	
Step	**More Information**

Configure the ASP.NET Web application to use Windows authentication (for IIS Basic)
—or—
Configure ASP.NET to use Forms authentication
	Edit Web.config in the Web service's virtual root directory
	Set the <**authentication**> element to either:
	`<authentication mode="Windows" />`
	—or—
	`<authentication mode="Forms" />`

Reset the password of the ASPNET account (used to run ASP.NET) to a known strong password	This allows you to create a duplicate local account (with the same user name and password) on the database server. This is required to enable the ASPNET account to respond to network authentication challenges from the database server, when it connects using Windows authentication.
	
	An alternative here is to use a least privileged domain account (if Windows authentication is permitted through the firewall).
	
	For more information, see "Process Identity for ASP.NET" in Chapter 8, ASP.NET Security.
	
	Edit Machine.config located in %windir%\Microsoft.NET\Framework\v1.0.3705\CONFIG
	
	Set your custom account username and password attributes on the <**processModel**> element
	Default
	
	`<!-- userName="machine" password="AutoGenerate"`
	`-->`
	
	Becomes
	
	`<!-- userName="machine"`
	`password="YourStrongPassword" -->`

Configuring SQL Server

Step	More Information
Create a Windows account on the SQL Server computer that matches the ASP.NET process account used to run the Web service (by default ASPNET)	The user name and password must match your ASP.NET process account. Give the account the following privileges: • Access this computer from the network • Deny logon locally • Log on as a batch job
Configure SQL Server for Windows authentication	
Create a SQL Server Login for the ASPNET account	This grants access to the SQL Server.
Create a new database user and map the login name to the database user	This grants access to the specified database.
Create a new user-defined database role within the database and place the database user into the role	
Establish database permissions for the database role	Grant minimum permissions See Chapter 12, Data Access Security.

Configuring secure communication

Step	More Information

Configure the Web site on the Web server for SSL	See How To: Set Up SSL on a Web Server in the Reference section of this guide.
Configure IPSec between Web server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Analysis

- ASP.NET on the Web server is running as a least privileged local account (the default ASPNET account), so potential damage from compromise is mitigated.
- SSL is used between browser and Web application to protect the Forms or Basic authentication credentials (both passed in clear text, although Basic uses Base64 encoding). SSL also protects the application-specific data returned from the Web application.
- For Forms authentication, SSL is used on all pages (not just the logon page) to protect the authentication cookie passed on all subsequent Web requests after the initial authentication.
- If SSL is used only on the initial logon page to encrypt the credentials passed for authentication, you should ensure that the Forms authentication ticket (contained within a cookie) is protected, because it is passed between client and server on each subsequent Web request. To encrypt the Forms authentication ticket, configure the **protection** attribute of the <**forms**> element as shown below and use the **Encrypt** method of the **FormsAuthentication** class to encrypt the ticket.

```
<authentication mode="Forms">
  <forms name="MyAppFormsAuth"
       loginUrl="login.aspx"
       protection="All"
       timeout="20"
       path="/" >
  </forms>
</authentication>
```

The **protection="All"** attribute specifies that when the application calls **FormsAuthentication.Encrypt**, the ticket should be validated (integrity checked) and encrypted. Call this method when you create the authentication ticket, typically within the application's Login button event handler.

```
string encryptedTicket = FormsAuthentication.Encrypt(authTicket);
```

For more information about Forms authentication and ticket encryption, see Chapter 8, ASP.NET Security.

- Similarly, SSL is used on all pages for Basic authentication because the Basic credentials are passed on all Web page requests and not just the initial one where the Basic credentials are supplied by the user.
- For Basic authentication, ASP.NET automatically creates a **WindowsPrincipal** object to represent the authenticated caller and associates it with the current Web request (**HttpContext.User**) where it is used by .NET authorization including **PrincipalPermission** demands and .NET roles.
- For Forms authentication, you must develop code to validate the supplied credentials against Active Directory and construct a **GenericPrincipal** to represent the authenticated user.

- Windows authentication to SQL Server means you avoid storing credentials on the Web server and it also means that credentials are not sent across the internal network to the SQL Server computer.
- IPSec between the Web service and database protects the data passed to and from the database on the corporate network.

Pitfalls

- The use of a duplicated local Windows account on the database server (one that matches the ASP.NET process account local to IIS) results in increased administration. Passwords must be manually updated and synchronized on a periodic basis.
- Basic authentication results in a pop-up dialog within the browser. To provide a more seamless logon experience, use Forms authentication.

Related scenarios

No connectivity from extranet to corporate network

For additional security, the extranet application can be built to require no connectivity back into the corporate network. In this scenario:

- A separate SQL Server database is located in the extranet and replication of data occurs from the internal database to the extranet database.
- Routers are used to refuse connections from the extranet to the corporate network. Connections can be established the other way using specific high ports.
- Connections from the corporate network to the extranet should always be performed through a dedicated server that has strong auditing and logging and through which users must authenticate before accessing the extranet.

More information

- See the following Microsoft TechNet articles:
  - Deploying SharePoint Portal Server in an Extranet Environment
- For more information about using Forms authentication with Active Directory, see How To: Use Forms Authentication with Active Directory in the Reference section of this guide.

Summary

This chapter has described how to secure two common extranet application scenarios.

For intranet and Internet application scenarios, see Chapter 5, Intranet Security, and Chapter 7, Internet Security.

Internet Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter describes how to secure common Internet application scenarios. It presents the characteristics of each scenario and describes the steps necessary to secure the scenario. Analysis sections are also included to provide further information. (17 printed pages)

Contents

Internet applications have large audiences, many potential uses, and varied security requirements. They range from portal applications that require no user authentication, through Web applications that provide content for registered users, to large-scale e-commerce applications that require full authentication, authorization, credit card validation and secure communication of sensitive data over public and internal networks.

As Internet application developers, you face a challenge to ensure that your application uses appropriate defense mechanisms and is designed to be scalable, high performance, and secure. Some of the challenges you face include:

- Choosing an appropriate user credential store, for example, a custom database or Microsoft® Active Directory® directory service
- Making your application work through firewalls
- Flowing security credentials across the multiple tiers of your application
- Performing authorization
- Ensuring the integrity and privacy of data as it flows across public and internal networks
- Securing your application's state with a database
- Ensuring the integrity of your application's data
- Implementing a solution that can scale to potentially huge numbers of users

The two common Internet application scenarios presented in this chapter, which are used to illustrate recommended authentication, authorization and secure communication techniques are:

- ASP.NET to SQL Server
- ASP.NET to Remote Enterprise Services to SQL Server

ASP.NET to SQL Server

In this scenario with two physical tiers, registered users securely log in to the Web-based application using a Web browser. The ASP.NET-based Web application makes secure connections to a Microsoft® SQL Server™ database to manage predominantly data retrieval tasks. An example is a portal application that provides news content to registered subscribers. This is shown in Figure 7.1.

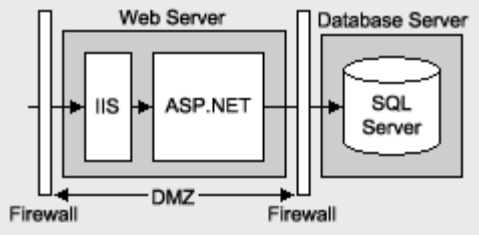

Figure 7.1. An ASP.NET Web application to SQL Server Internet scenario

Characteristics

This scenario has the following characteristics:

- Users have a number of different browser types.
- Anonymous users can browse the application's unrestricted pages.
- Users must register or log on (through an HTML form) before being allowed to view restricted pages.
- User credentials are validated against a SQL Server database.
- All user input (such as user credentials) that is used in database queries is validated to mitigate the threat of SQL injection attacks.
- The front-end Web application is located within a perimeter network (also known as DMZ, demilitarized zone, and screened subnet), with firewalls separating it from the Internet and the internal corporate network (and the SQL Server database).
- The application requires strong security, high levels of scalability, and detailed auditing.
- The database trusts the application to authenticate users properly (that is, the application makes calls to the database on behalf of the users).
- The Web application connects to the database by using the ASP.NET process account.
- A single user-defined database role is used within SQL Server for database authorization.

Secure the Scenario

In this scenario, the Web application presents a logon page to accept credentials. Successfully validated users are allowed to proceed, all others are denied access. The database authenticates against the ASP.NET default process identity, which is a least privileged account (that is, the database trusts the ASP.NET application).

Table 7.1. Security summary

Category	Detail
Authentication	IIS is configured to allow anonymous access; the ASP.NET Web application authenticates users with Forms authentication to acquire credentials. Validation is against a SQL Server database. Users' passwords are not stored in the database. Instead password hashes with salt values are stored. The salt mitigates the threat associated with dictionary attacks. Microsoft® Windows® authentication is used to connect to the database using the least privileged Windows account used to run the ASP.NET Web application.
Authorization	The ASP.NET process account is authorized to access system resources on the Web server. Resources are protected with Windows ACLs.

	Access to the database is authorized using the ASP.NET application identity.
Secure Communication	Secure sensitive data sent between the users and the Web application by using SSL. Secure sensitive data sent between the Web server and the database server by using IPSec.

The Result

Figure 7.2 shows the recommended security configuration for this scenario.

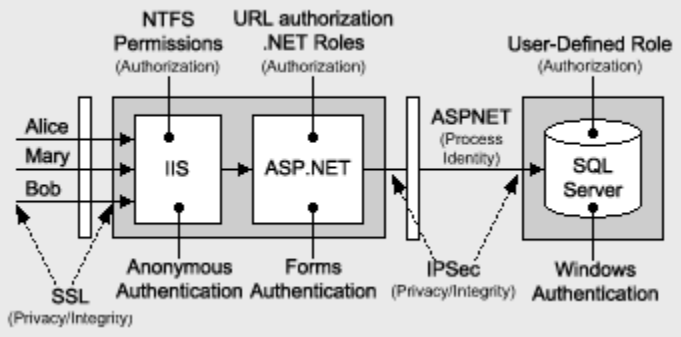

Figure 7.2. The recommended security configuration for the ASP.NET to SQL Server Internet scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Creating custom ASP.NET accounts (see How To: Create a Custom Account to Run ASP.NET in the Reference section of this guide)
- Creating a least privileged database account (see Chapter 12, Data Access Security)
- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)

Configure the Web server

Configure IIS	
Step	**More Information**
Enable Anonymous access for your Web application's virtual root directory	To work with IIS authentication settings, use the IIS MMC snap-in. Right-click your application's virtual directory, and then click **Properties**. Click the **Directory Security** tab, and then click **Edit** within the **Anonymous access and authentication control** group.
Configure ASP.NET	
Step	**More Information**
Reset the password of the ASPNET account (used to run ASP.NET) to a known strong password	This allows you to create a duplicate local account (with the same user name and password) on the database server. This is required to allow the ASPNET account to respond to network authentication challenges from the database

	server when it connects using Windows authentication.
	
	An alternative here is to use a least privileged domain account (if Windows authentication is permitted through the firewall).
	
	For more information, see Process Identity for ASP.NET in Chapter 8, "ASP.NET Security."
	
	Edit Machine.config located in %windir%\Microsoft.NET\Framework\v1.0.3705\CONFIG
	
	Set your custom account user name and password attributes on the <**processModel**> element
	Default
	
	`<!-- userName="machine" password="AutoGenerate" -- >`
	
	Becomes
	
	`<!-- userName="machine" password="YourStrongPassword" -- >`

Configure your ASP.NET Web application to use Forms authentication (with SSL)	Edit Web.config in your application's virtual directory root
	
	Set the <**authentication**> element to:
	
	`<authentication mode="Forms" >`
	`<forms name="MyAppFormsAuth"`
	`loginUrl="login.aspx"`
	`protection="All"`
	`timeout="20"`
	`path="/" >`
	`</forms>`
	`</authentication>`
	
	For more information about using Forms authentication against a SQL Server database, see How To: Use Forms Authentication with SQL Server 2000 in the Reference section of this guide.

Configuring SQL Server

Step	More Information
Create a Windows account on your SQL Server computer that matches the ASP.NET process account	The user name and password must match your custom ASP.NET application account or must be ASPNET if you are using the default account.
Configure SQL Server for Windows	

authentication	
Create a SQL Server Login for your custom ASP.NET application account	This grants access to SQL Server.
Create a new database user and map the login name to the database user	This grants access to the specified database.
Create a new user-defined database role within the database and place the database user into the role	
Establish database permissions for the database role	Grant minimum permissions. For more information, see Chapter 12, Data Access Security.

Configuring Secure Communication

Step	More Information
Configure the Web site for SSL	See How To: Setup SSL on a Web Server in the Reference section of this guide.
Configure IPSec between application server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Analysis

- Forms authentication is ideal in this scenario because the users do not have Windows accounts. The Forms login page is used to acquire user credentials. Credential validation must be performed by application code. Any data store can be used. A SQL Server database is the most common solution, although Active Directory provides an alternate credential store.
- With Forms authentication, you must protect the initial logon credentials with SSL. The Forms authentication ticket (passed as a cookie on subsequent Web requests from the authenticated client) must also be protected. You could use SSL for all pages in order to protect the ticket, or alternatively you can encrypt the Forms authentication ticket by configuring the **protection** attribute of the <**forms**> element (to **All** or **Encrypt**) and use the **Encrypt** method of the **FormsAuthentication** class to encrypt the ticket.

  The **protection="All"** attribute specifies that when the application calls **FormsAuthentication.Encrypt**, the ticket should be validated (integrity checked) and encrypted. Call this method when you create the authentication ticket, typically within the application's Login button event handler.

  ```
  string encryptedTicket = FormsAuthentication.Encrypt(authTicket);
  ```

  For more information about Forms authentication and ticket encryption, see Chapter 8, ASP.NET Security.

- ASP.NET runs as the least privileged local ASPNET account, so potential damage from compromise is mitigated.
- URL authorization on the Web server allows unauthenticated users to browse unrestricted Web pages and forces authentication for restricted pages.
- Because impersonation is not enabled, any local or remote resource access performed by the Web-based application is performed using the ASPNET account security context. Windows ACLs on secure resources should be set accordingly.
- User credentials are validated against a custom SQL Server database. Password hashes (with salt) are stored within the database. For more information, see Authenticating Users against a Database in Chapter 12, "Data Access Security."
- By using Windows authentication to SQL Server, you avoid storing credentials in files on the Web server and also passing them over the network.

- If your application currently uses SQL authentication, you must securely store the database connection string as it contains user names and passwords. Consider using DPAPI. For more details, see Storing Database Connection Strings Securely, in Chapter 12, "Data Access Security."
- The use of a duplicated Windows account on the database server (one that matches the ASP.NET process account) results in increased administration. If a password is changed on one computer, it must be synchronized and updated on all computers. In some scenarios, you may be able to use a least-privileged domain account for easier administration.
- IPSec between the Web server and database server ensures the privacy of the data sent to and from the database.
- SSL between browser and Web server protects credentials and any other security sensitive data such as credit card numbers.
- If you use a Web farm, ensure that the encryption keys, for example those used to encrypt the Forms authentication ticket (and specified by the <**machineKey**> element in Machine.config), are consistent across all servers in the farm. See Chapter 8, ASP.NET Security, for further details about using ASP.NET in a Web farm scenario.

Pitfalls

The application must flow the original caller's identity to the database to support auditing requirements. Caller identity may be passed using stored procedure parameters.

Related Scenarios

Forms authentication against Active Directory

The user credentials that are accepted from the Forms login page can be authenticated against various stores. Active Directory is an alternate to using a SQL Server database.

More information

For more information, see How To: Use Forms Authentication with Active Directory in the Reference section of this guide.

.NET roles for authorization

The preceding scenario doesn't take into consideration the different types of users accessing the application. For example, a portal server could have different subscription levels such as Standard, Premier, and Enterprise.

If role information is maintained in the user store (SQL Server database), the application can create a **GenericPrincipal** object in which role and identity information can be stored. After the **GenericPrincipal** is created and added to the Web request context (using **HttpContext.User**), you can add programmatic role checks to method code or you can decorate methods and pages with **PrincipalPermission** attributes to demand role membership.

More information

- For more information about creating **GenericPrincipal** objects that contain role lists, see How To: Create GenericPrincipal Objects with Forms Authentication in the Reference section of this guide.
- For more information about **PrincipalPermission** demands and programmatic role checks, see Chapter 8, ASP.NET Security.

Using a domain anonymous account at the Web server

In this scenario variation, the default anonymous Internet user account (a local account called IUSR_MACHINE) is replaced by a domain account. The domain account is configured with the minimum privileges necessary to run the application (you can start with no privilege and incrementally add privileges). If you have multiple Web-based applications, you can use different domain accounts (one for each Web-based application or virtual directory).

In order to flow the security context of the anonymous domain account from IIS to ASP.NET, turn on impersonation for the Web-based application by using the following web.config file setting:

```
<identity impersonate="true" />
```

If the Web-based application communicates with a remote resource such as a database, the domain account must be granted the necessary permissions to the resource. For example, if the application accesses a remote file system, ACLs must be configured appropriately to give (at minimum) read access to the domain account. If the application accesses a SQL Server database, the domain account must be mapped using a SQL login to a database login.

As the security context that flows through the application is that of the anonymous account, the original caller's identity (captured through Forms authentication) must be passed at the application level from tier to tier; for example, through method and stored procedure parameters.

More information

- For more information regarding this approach, see Using the Anonymous Internet User Account within Chapter 8, "ASP.NET Security."
- Before implementing this scenario, see article Q259353, Must Enter Password Manually After You Set Password Synchronization in the Microsoft Knowledge Base.

ASP.NET to Remote Enterprise Services to SQL Server

In this scenario, a Web server running ASP.NET pages makes secure connections to serviced components, located on a remote application server that in turn connects to a SQL Server database. In common with many Internet application infrastructures, the Web server and application server are separated by a firewall (and the Web server is located within a perimeter network). Serviced components make secure connections to SQL Server.

As an example, consider an Internet banking application that provides sensitive data, (for example, private financial details) to users. All banking transactions from the client to the database must be secured and data integrity is critical. Not only does the traffic to and from the user need to be secured but the traffic to and from the database needs to be secured as well. This is shown in Figure 7.3.

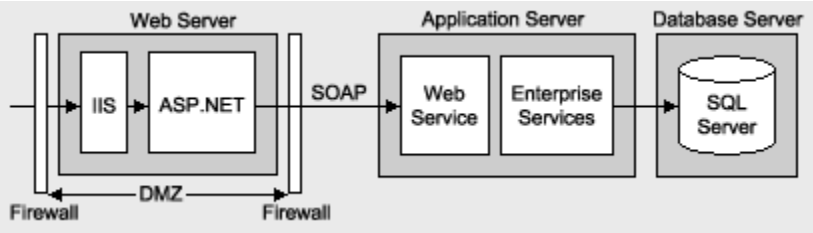

Figure 7.3. An ASP.NET to remote Enterprise Services to SQL Server Internet scenario

Characteristics

- Users have a number of different browser types.
- Anonymous users can browse the application's unrestricted pages.
- Users must register or log on (through an HTML form) before being allowed to view restricted pages.
- The front-end Web-based application is located within a perimeter network, with firewalls separating it from the Internet and the internal corporate network (and the application server).
- The application requires strong security, high levels of scalability, and detailed auditing.
- The Web-based application uses SOAP to connect to a Web services layer, which provides an interface to the serviced components that run within an Enterprise Services application on the application server. SOAP is preferred to DCOM due to firewall restrictions.

- SQL Server is using a single user-defined database role for authorization.
- Data is security sensitive and integrity and privacy must be secured over the network and in all persistent data stores.
- Enterprise Services (COM+) transactions are used to enforce data integrity.

Secure the Scenario

In this scenario, the Web service accepts credentials from a Forms login page and then authenticates the caller against a SQL Server database. The login page uses SSL to protect the user's credentials passed over the Internet.

The Web-based application communicates with a Web service, which provides an interface to the business services implemented within serviced components. The Web service trusts the Web-based application (inside the perimeter network) and authenticates the ASP.NET process identity. The user's identity is passed through all tiers at the application level using method and stored procedure parameters. This information is used for auditing the users' actions across the tiers.

Table 7.2. Security measures

Category	Detail
Authentication	Provide strong authentication at the Web server. Authenticate the Enterprise Services application identity at the database. IIS is configured for anonymous access and the Web-based application authenticates users with Forms authentication (against a SQL Server database). The Web service's virtual directory is configured for Integrated Windows authentication. Web services authenticate the Web-based application's process identity. Windows authentication is used to connect to the database. The database authenticates the least privileged Windows account used to run the Enterprise Services application.
Authorization	The trusted subsystem model is used and per-user authorization occurs only within the Web application. User access to pages on the Web server is controlled with URL authorization. The ASP.NET process account is authorized to access system resources on the Web server. Resources are protected with ACLs. Permissions within the database are controlled by a user-defined role. The Enterprise Services application identity is a member of the role. The Enterprise Services process account is authorized to access system resources on the application server. Resources are protected ACLs.
Secure Communication	Sensitive data sent between the users and the Web-based application is secured with SSL. Sensitive data sent between the Web server and Web service is secured with SSL. Sensitive data sent between serviced components and the database is secured with IPSec.

The Result

Figure 7.4 shows the recommended security configuration for this scenario.

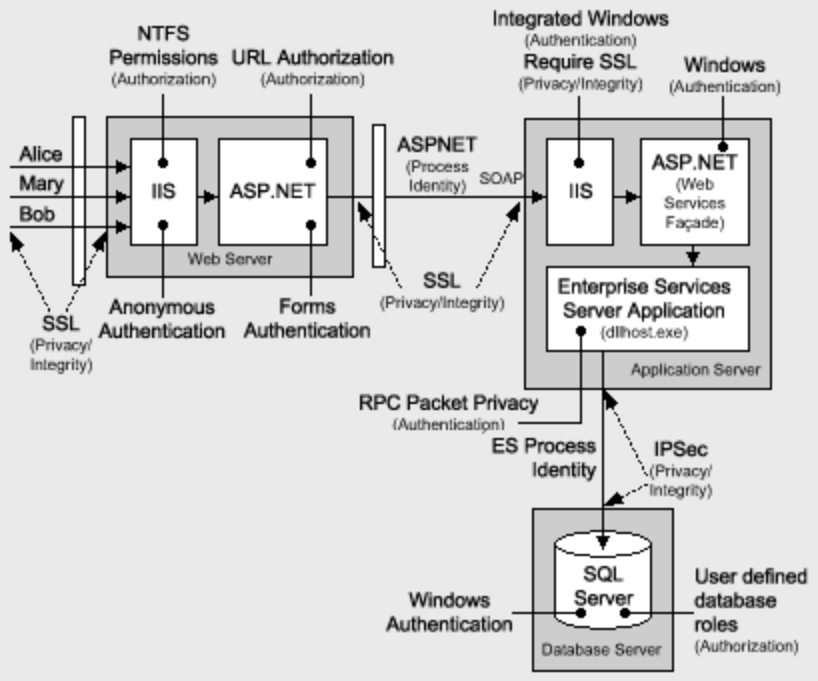

Figure 7.4. The recommended security configuration for the ASP.NET to remote Enterprise Services to SQL Server Internet scenario

Security Configuration Steps

Before you begin, you'll want to see the following:

- Creating a least privileged database account (see Chapter 12, Data Access Security)
- Configuring SSL on a Web server (see How To: Set Up SSL on a Web Server in the Reference section of this guide)
- Configuring IPSec (see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide)
- Configuring Enterprise Services security (see How To: Use Role-based Security with Enterprise Services in the Reference section of this guide)

Configure the Web server

Configure IIS	
Step	**More Information**
Enable Anonymous access for your Web-based application's virtual root directory	
Configure ASP.NET	
Step	**More Information**
Reset the password of the ASPNET account (used to run ASP.NET) to a known strong password	This allows you to create a duplicate local account (with the same user name and password) on the application server. This is required to enable the ASPNET account to respond to network authentication challenges from the

	application server.
	
	An alternative is to use a least privileged domain account (if Windows authentication is permitted through the firewall).
	
	For more information, see Process Identity for ASP.NET in Chapter 8, "ASP.NET Security."
	
	Edit Machine.config located in %windir%\Microsoft.NET\Framework\v1.0.3705\CONFIG
	
	Set your custom account username and password attributes on the <**processModel**> element.
	Default
	
	`<!-- userName="machine" password="AutoGenerate" -- >`
	
	Becomes
	
	`<!-- userName="machine" password="YourStrongPassword" -- >`

Configure your Web-based application to use Forms authentication (with SSL)	Edit Web.config in your application's virtual directory root
	Set the <**authentication**> element to:
	
	`<authentication mode="Forms" >`
	`  <forms name="MyAppFormsAuth"`
	`  loginUrl="login.aspx"`
	`  protection="All"`
	`  timeout="20"`
	`  path="/" >`
	`  </forms>`
	`</authentication>`
	
	For more information about using Forms authentication against a SQL Server database, see How To: Use Forms Authentication with SQL Server 2000 in the Reference section of this guide.

Configure the application server

Configure IIS	
Step	**More Information**
Disable anonymous access	
Configure Integrated Windows authentication	IIS authenticates the ASP.NET process identity from the Web-based application on the Web server.

Configure ASP.NET	
Step	**More Information**
Use Windows authentication	Edit Web.config in your Web service's virtual directory root. Set the <**authentication**> element to: `<authentication mode="Windows" />`

Configure Enterprise Services	
Step	**More Information**
Create a least privileged custom account for running the Enterprise Services server application	**Note:** If you use a local account, you must also create a duplicate account on the database server computer.
Configure the Enterprise Services application to use the custom account	Refer to Configuring Security within Chapter 9, "Enterprise Services Security."
Enable role-based access checking	Refer to Configuring Security within Chapter 9, "Enterprise Services Security."
Add a single Enterprise Services (COM+) role to the application called (for example Trusted Web Service)	Full end-user authorization is performed by the Web-based application. The Web service (and serviced components) only allows access to members of the Trusted Web Service role.
Add the local ASPNET account to the Trusted Web Service role	Refer to Configuring Security within Chapter 9, "Enterprise Services Security."

Configuring SQL Server

Step	More Information
Create a Windows account on your SQL Server computer that matches the Enterprise Services application account	The user name and password must match your custom Enterprise Services account.
Configure SQL Server for Windows authentication	
Create a SQL Server Login for your custom Enterprise Services account	This grants access to the SQL Server.
Create a new database user and map the login name to the database user	This grants access to the specified database.
Create a new user-defined database role and add the database user to the role	
Establish database permissions for the database role	Grant minimum permissions For details, see Chapter 12, Data Access Security.

Configuring secure communication

Step	More Information
Configure the Web site for SSL	See How To: Setup SSL on a Web Server in the Reference section of this guide.
Configure SSL between the Web server and application server.	See How To: Call a Web Service Using SSL in the Reference section of this guide.

Configure IPSec between application server and database server	See How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Analysis

- Forms authentication is ideal in this scenario because the users do not have Windows accounts. The Forms login page is used to acquire user credentials. Credential validation must be performed by application code. Any data store can be used. A SQL Server database is the most common solution, although Active Directory provides an alternate credential store.
- The Web-based application is running as the least privileged local ASPNET account, so potential damage from compromise is mitigated.
- URL authorization on the Web server allows unauthenticated users to browse unrestricted Web pages and forces authentication for restricted pages.
- Because impersonation is not enabled, any local or remote resource access performed by the Web-based application does so using the ASPNET account security context. ACLs should be configured accordingly.
- User credentials are validated against a custom SQL Server database. Password hashes (with salt) are stored within the database. For more information, see Authenticating Users against a Database in Chapter 12, "Data Access Security."
- Windows authentication to SQL Server means you avoid storing credentials in files on the application server and avoid passing them across the network.
- The use of a duplicated Windows account on the database server (one that matches the Enterprise Services process account) results in increased administration. If a password is changed on one computer, it must be synchronized and updated on all computers. In some scenarios, you may be able to use a least-privileged domain account for easier administration.
- When the Web application calls the Web service, it must configure the Web service proxy using DefaultCredentials (that is, the ASP.NET process account; ASPNET).

```
proxy.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

  For more information, see Passing Credentials For Authentication to Web Services in Chapter 10, "Web Services Security."

- SSL between the Web server and Web service layer (that fronts the serviced components on the application server) ensures the privacy of the data sent between the two servers.
- The Enterprise Services application is configured for application-level role-based security. The configuration permits only the local ASPNET account (used to run the Web service) to access the serviced components.
- IPSec between the application server and database server ensures the privacy of the data sent to and from the database.
- SSL between browser and Web server protects credentials and bank account details.

Pitfalls

The application must flow the original caller's identity to the database to support auditing requirements. Caller identity may be passed using stored procedure parameters.

Related Scenarios

Forms authentication against Active Directory

The user credentials that are accepted from the Forms login page can be authenticated against various stores. Active Directory is an alternate to using a SQL Server database.

More information

For more information, see How To: Use Forms Authentication with Active Directory in the Reference section of this guide.

Using DCOM

Windows 2000 (SP3 or SP2 with QFE 18.1) or Windows Server allows you to configure Enterprise Services applications to use a static endpoint. If a firewall separates the client from the server, this means that you need to open only two ports in the firewall. Specifically, you must open port 135 for RPC and a port for your Enterprise Services application.

This enhancement to DCOM makes it a valid choice of communication protocol between Web server and application server and removes the requirement to have a Web services layer.

Important If your application requires distributed transactions to flow between the two servers, DCOM must be used. Transactions cannot flow over SOAP. In the SOAP scenario, transactions must be initiated by the serviced components on the application server.

More information

For more information, see Chapter 9, Enterprise Services Security.

Using .NET Remoting

Remoting can be a valid choice when you don't need services provided by Enterprise Services such as transactions, queued components, object pooling, and so on. .NET Remoting solutions also support network load balancing at the middle tier. Note the following when you use .NET Remoting:

- For ultimate performance, use the TCP channel and host in a Windows service. Note that this channel provides no authentication and authorization mechanism by default. The TCP channel is designed for trusted subsystem scenarios. You can use an IPSec policy to establish a secure channel and to ensure that only the Web server communicates with the application server.
- If you need authentication and authorization checks using **IPrincipal** objects, you should host the remote objects in ASP.NET and use the HTTP channel. This allows you use the IIS and ASP.NET security features.
- The remote object can connect to the database using Windows authentication and can use the host process identity (either ASP.NET or a Windows service identity).

More information

For more information about .NET Remoting security, see Chapter 11, .NET Remoting Security.

Summary

This chapter has described how to secure a set of common Internet application scenarios.

For Intranet and extranet application scenarios, see Chapter 5, "Intranet Security," and Chapter 6, Extranet Security.

ASP.NET Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter presents guidance and recommendations that will help you build secure ASP.NET Web applications. Much of the guidance and many of the recommendations presented in this chapter also apply to the development of ASP.NET Web services and .NET Remoting objects hosted by ASP.NET.

Contents

ASP.NET Security Architecture

ASP.NET works in conjunction with IIS, the .NET Framework, and the underlying security services provided by the operating system, to provide a range of authentication and authorization mechanisms. These are summarized in Figure 8.1.

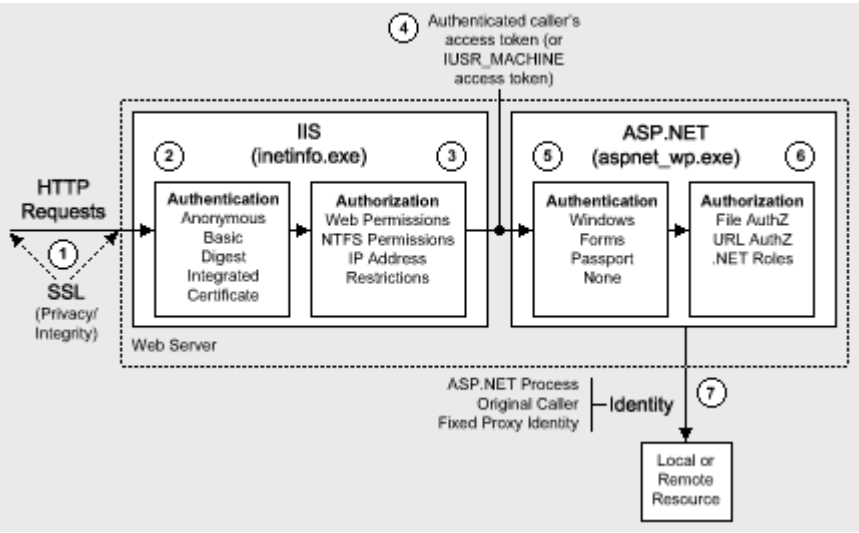

Figure 8.1. ASP.NET security services

Figure 8.1 illustrates the authentication and authorization mechanisms provided by IIS and ASP.NET. When a client issues a Web request, the following sequence of authentication and authorization events occurs:

1. The HTTP(S) Web request is received from the network. SSL can be used to ensure the server identity (using server certificates) and, optionally, the client identity.

   Note SSL also provides a secure channel to protect sensitive data passed between client and server (and vice-versa).

2. IIS authenticates the caller by using Basic, Digest, Integrated (NTLM or Kerberos), or Certificate authentication. If all or part of your site does not require authenticated access, IIS can be configured for anonymous authentication. IIS creates a Windows access token for each authenticated user. If anonymous authentication is selected, IIS creates an access token for the anonymous Internet user account (which, by default, is IUSR_MACHINE).

3. IIS authorizes the caller to access the requested resource. NTFS permissions defined by ACLs attached to the requested resource are used to authorize access. IIS can also be configured to accept requests only from client computers with specific IP addresses.

4. IIS passes the authenticated caller's Windows access token to ASP.NET (this may be the anonymous Internet user's access token, if anonymous authentication is being used).

5. ASP.NET authenticates the caller.

   If ASP.NET is configured for Windows authentication, no additional authentication occurs at this point. ASP.NET will accept any token it receives from IIS.

   If ASP.NET is configured for Forms authentication, the credentials supplied by the caller (using an HTML form) are authenticated against a data store; typically a Microsoft® SQL Server™ database or Microsoft Active Directory® directory service. If ASP.NET is configured for Passport authentication, the user is redirected to a Passport site and the Passport authentication service authenticates the user.

6. ASP.NET authorizes access to the requested resource or operation.

   The **UrlAuthorizationModule** (a system provided HTTP module) uses authorization rules configured in Web.config (specifically, the <**authorization**> element) to ensure that the caller can access the requested file or folder.

   With Windows authentication, the **FileAuthorizationModule** (another HTTP module) checks that the caller has the necessary permission to access the requested resource. The caller's access token is compared against the ACL that protects the resource.

   .NET roles can also be used (either declaratively or programmatically) to ensure that the caller is authorized to access the requested resource or perform the requested operation.

7. Code within your application accesses local and/or remote resources by using a particular identity. By default, ASP.NET performs no impersonation and as a result, the configured ASP.NET process account provides the identity. Alternate options include the original caller's identity (if impersonation is enabled), or a configured service identity.

Gatekeepers

The authorization points (or gatekeepers) within an ASP.NET Web application are provided by IIS and ASP.NET:

IIS

With anonymous authentication turned off, IIS permits requests only from users that it can authenticate either in its domain or in a trusted domain.

For static file types (for example .jpg, .gif and .htm files—files that are not mapped to an ISAPI extension), IIS uses the NTFS permissions associated with the requested file to perform access control.

ASP.NET

The ASP.NET gatekeepers include the **UrlAuthorizationModule**, **FileAuthorizationModule** and Principal permission demands and role checks.

UrlAuthorizationModule

You can configure <**authorization**> elements within your application's Web.config file to control which users and groups of users should have access to the application. Authorization is based on the **IPrincipal** object stored in **HttpContext.User**.

FileAuthorizationModule

For file types mapped by IIS to the ASP.NET ISAPI extension (Aspnet_isapi.dll), automatic access checks are performed using the authenticated user's Windows access token (which may be IUSR_MACHINE) against the ACL attached to the requested ASP.NET file.

Note Impersonation is not required for file authorization to work.

The **FileAuthorizationModule** class only performs access checks against the requested file, and not for files accessed by the code in the requested page, although these are access checked by IIS.

For example, if you request Default.aspx and it contains an embedded user control (Usercontrol.ascx), which in turn includes an image tag (pointing to Image.gif), the **FileAuthorizationModule** performs an access check for Default.aspx and Usercontrol.ascx, because these file types are mapped by IIS to the ASP.NET ISAPI extension.

The **FileAuthorizationModule** does not perform a check for Image.gif, because this is a static file handled internally by IIS. However, as access checks for static files are performed by IIS, the authenticated user must still be granted read permission to the file with an appropriately configured ACL.

This scenario is shown in Figure 8.2.

Note to system administrators: The authenticated user requires NTFS read permissions to all of the files involved in the scenario. The only variable is regarding which gatekeeper is used to enforce access control. The ASP.NET process account only requires read access to the ASP.NET registered file types.

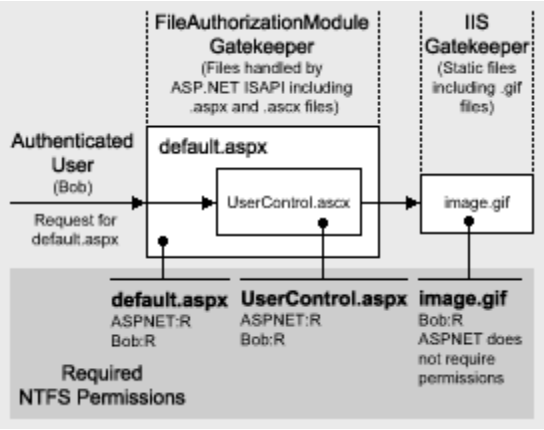

Figure 8.2. IIS and ASP.NET gatekeepers working together

In this scenario you can prevent access at the file gate. If you configure the ACL attached to Default.aspx and deny access to a particular user, the user control or any embedded images will not get a chance to be sent to the client by the code in Default.aspx. If the user requests the images directly, IIS performs the access checks itself.

Principal permission demands and explicit role checks

In addition to the IIS and ASP.NET configurable gatekeepers, you can also use principal permission demands (declaratively or programmatically) as an additional fine-grained access control mechanism. Principal permission checks (performed by the **PrincipalPermissionAttribute** class) allow you to control access to classes, methods,

or individual code blocks based on the identity and group membership of individual users, as defined by the **IPrincipal** object attached to the current thread.

Note Principal permission demands used to demand role membership are different from calling **IPrincipal.IsInRole** to test role membership; the former results in an exception if the caller is not a member of the specified role, while the latter simply returns a Boolean value to confirm role membership.

With Windows authentication, ASP.NET automatically attaches a **WindowsPrincipal** object that represents the authenticated user to the current Web request (using **HttpContext.User**). Forms and Passport authentication create a **GenericPrincipal** object with the appropriate identity and no roles and attaches it to the **HttpContext.User**.

More information

- For more information about configuring security, see Configuring Security later in this chapter.
- For more information about programming security (and **IPrincipal** objects), see Programming Security later in this chapter.

Authentication and Authorization Strategies

ASP.NET provides a number of declarative and programmatic authorization mechanisms that can be used in conjunction with a variety of authentication schemes. This allows you to develop an in depth authorization strategy and one that can be configured to provide varying degrees of granularity; for example, per-user or per-user group (role-based).

This section shows you which authorization options (both configurable and programmatic) are available for a set of commonly used authentication options.

The authentication options that follow are summarized here:

- Windows authentication with impersonation
- Windows authentication without impersonation
- Windows authentication using a fixed identity
- Forms authentication
- Passport authentication

Available Authorization Options

The following table shows you the set of available authorization options. For each one the table indicates whether or not Windows authentication and/or impersonation are required. If Windows authentication is not required, the particular authorization option is available for all other authentication types. Use the table to help refine your authentication/authorization strategy.

Table 8.1. Windows authentication and impersonation requirements

Authorization Option	Requires Windows Authentication	Requires Impersonation
FileAuthorizationModule	Yes	No
UrlAuthorizationModule	No	No
Principal Permission Demands	No	No
.NET Roles	No	No
Enterprise Services Roles	Yes	Yes (within the ASP.NET Web application)
NTFS Permissions (for directly	N/A–These files are not handled by	No (IIS performs the access check.)

requested static files types; not mapped to an ISAPI extension)	ASP.NET. With any (non-Anonymous) IIS authentication mechanism, permissions should be configured for individual authenticated users. With Anonymous authentication, permissions should be configured for IUSR_MACHINE.	
NTFS Permissions (for files accessed by Web application code)	No	No If impersonating, configure ACLs against the impersonated Windows identity, which is either the original caller or the identity specified on the <**identity**> element in Web.config*.

* The impersonated identity may be the original caller or the identity specified on the <**identity**> element in Web.config. Consider the following two <**identity**> elements.

```
<identity impersonate="true" />

<identity impersonate="true" userName="Bob" password="pwd" />
```

The first configuration results in the impersonation of the original caller (as authenticated by IIS), while the second results in the identity Bob. The second configuration is not recommended for two reasons:

- It requires that you grant the ASP.NET process identity the "Act as part of the operating system" privilege on the Microsoft Windows® 2000 operating system.
- It also requires you to include a plain text password in Web.config.

Both of these restrictions will be lifted in the next release of the .NET Framework.

Windows Authentication with Impersonation

The following configuration elements show you how to enable Windows (IIS) authentication and impersonation declaratively in Web.config or Machine.config.

Note You should configure authentication on a per-application basis in each application's Web.config file.

```
<authentication mode="Windows" />

<identity impersonate="true" />
```

With this configuration, your ASP.NET application code impersonates the IIS-authenticated caller.

Configurable security

When you use Windows authentication together with impersonation, the following authorization options are available to you:

- **Windows ACLs**
  - **Client Requested Resources**. The ASP.NET **FileAuthorizationModule** performs access checks for requested file types that are mapped to the ASP.NET ISAPI. It uses the original caller's access token and ACL attached to requested resources in order to perform access checks.

    For static files types (not mapped to an ISAPI extension), IIS performs access checks using the caller's access token and ACL attached to the file.

- **Resources Accessed by Your Application**. You can configure Windows ACLs on resources accessed by your application (files, folders, registry keys, Active Directory objects, and so on) against the original caller.

- **URL Authorization**. Configure URL authorization in Web.config. With Windows authentication, user names take the form DomainName\UserName and roles map one-to-one with Windows groups.

```
<authorization>
  <deny user="DomainName\UserName" />
  <allow roles="DomainName\WindowsGroup" />
</authorization>
```

- **Enterprise Services (COM+) Roles**. Roles are maintained in the COM+ catalog. You can configure roles with the Component Services administration tool or script.

Programmatic security

Programmatic security refers to security checks located within your Web application code. The following programmatic security options are available when you use Windows authentication and impersonation:

- **PrincipalPermission Demands**
  - Imperative (in-line within a method's code)

```
PrincipalPermission permCheck = new PrincipalPermission(
                                        null,
                                          @"DomainName\
                                          WindowsGroup");
permCheck.Demand();
```

  - Declarative (attributes preceding interfaces, classes and methods)

```
[PrincipalPermission(SecurityAction.Demand,
                     Role=@"DomainName\WindowsGroup)]
```

- **Explicit Role Checks**. You can perform role checking using the **IPrincipal** interface.

```
IPrincipal.IsInRole(@"DomainName\WindowsGroup");
```

- **Enterprise Services (COM+) Roles**. You can perform role checking programmatically using the **ContextUtil** class.

```
ContextUtil.IsCallerInRole("Manager")
```

When to use

Use Windows authentication and impersonation when:

- Your application's users have Windows accounts that can be authenticated by the server.
- You need to flow the original caller's security context to the middle tier and/or data tier of your Web application to support fine-grained (per-user) authorization.
- You need to flow the original caller's security context to the downstream tiers to support operating system level auditing.

Before using impersonation within your application, make sure you understand the relative trade-offs of this approach in comparison to using the trusted subsystem model. These were elaborated upon in Choosing an Authentication Mechanism in Chapter 3, "Authentication and Authorization."

The disadvantages of impersonation include:

- Reduced application scalability due to the inability to effectively pool database connections.
- Increased administration effort as ACLs on back-end resources need to be configured for individual users.
- Delegation requires Kerberos authentication and a suitably configured environment.

More information

- For more information about Windows authentication, see "Windows Authentication" later in this chapter.
- For more information about impersonation, see "Impersonation" later in this chapter.
- For more information about URL authorization, see "URL Authorization Notes" later in this chapter.
- For more information about Enterprise Services (COM+) roles, see Chapter 9, Enterprise Services Security.
- For more information about **PrincipalPermission** demands, see Identities and Principals in Chapter 2, "Security Model for ASP.NET Application."

Windows Authentication without Impersonation

The following configuration elements show how you enable Windows (IIS) authentication with no impersonation declaratively in Web.config.

```
<authentication mode="Windows" />

<!-- The following setting is equivalent to having no identity

  element -->

<identity impersonate="false" />
```

Configurable security

When you use Windows authentication without impersonation, the following authorization options are available to you:

- **Windows ACLs**
  - **Client Requested Resources**. The ASP.NET **FileAuthorizationModule** performs access checks for requested file types that are mapped to the ASP.NET ISAPI. It uses the original caller's access token and ACL attached to requested resources in order to perform access checks. Impersonation is not required.

    For static files types (not mapped to an ISAPI extension) IIS performs access checks using the caller's access token and ACL attached to the file.

- **Resources accessed by your application**. Configure Windows ACLs on resources accessed by your application (files, folders, registry keys, Active Directory objects) against the ASP.NET process identity.

- **URL Authorization**. Configure URL Authorization in Web.config. With Windows authentication, user names take the form DomainName\UserName and roles map one-to-one with Windows groups.

```
<authorization>
  <deny user="DomainName\UserName" />
  <allow roles="DomainName\WindowsGroup" />
</authorization>
```

Programmatic security

The following programmatic security options are available:

- **Principal Permission Demands**
  - Imperative

```
PrincipalPermission permCheck = new PrincipalPermission(
                                        null,
  @"DomainName\WindowsGroup");
    permCheck.Demand();
```

  - Declarative

```
 [PrincipalPermission(SecurityAction.Demand,
                      Role=@"DomainName\WindowsGroup")]
```

- **Explicit Role Checks**. You can perform role checking using the **IPrincipal** interface.

```
IPrincipal.IsInRole(@"DomainName\WindowsGroup");
```

When to use

Use Windows authentication without impersonation when:

- Your application's users have Windows accounts that can be authenticated by the server.
- You want to use a fixed identity to access downstream resources (for example, databases) in order to support connection pooling.

More information

- For more information about Windows authentication, see "Windows Authentication" later in this chapter.
- For more information about URL authorization, see "URL Authorization Notes", later in this chapter.

- For more information about **PrincipalPermission** demands, see Identities and Principals in Chapter 2, "Security Model for ASP.NET Application."

Windows Authentication Using a Fixed Identity

The <**identity**> element in Web.config supports optional user name and password attributes, which allows you to configure a specific fixed identity for your application to impersonate. This is shown in the following configuration file fragment.

```
<identity impersonate="true" userName="DomainName\UserName"

                             password="ClearTextPassword" />
```

When to use

This approach is not recommended for the current version (version 1) of the .NET Framework in secure environments for two reasons:

- User names and passwords should not be stored in plain text in configuration files, particularly configuration files stored in virtual directories.
- On Windows 2000, this approach forces you to grant the ASP.NET process account the "Act as part of the operating system" privilege. This reduces the security of your Web application and increases the threat should an attacker compromise the Web application process.

The .NET Framework version 1.1 will provide an enhancement for this scenario on Windows 2000:

- The credentials will be encrypted.
- The log on will be performed by the IIS process, so that ASP.NET does not required the "Act as part of the operating system" privilege.

Forms Authentication

The following configuration elements show how you enable Forms authentication declaratively in Web.config.

```
<authentication mode="Forms">

  <forms loginUrl="logon.aspx" name="AuthCookie" timeout="60"

    path="/">

  </forms>

</authentication>
```

Configurable security

When you use Forms authentication, the following authorization options are available to you:

- **Windows ACLs**
  - **Client Requested Resources**. Requested resources require ACLs that allow read access to the anonymous Internet user account. (IIS should be configured to allow anonymous access when you use Forms authentication).

    ASP.NET File authorization is not available because it requires Windows authentication.

  - **Resources Accessed by Your Application**. Configure Windows ACLs on resources accessed by your application (files, folders, registry keys, and Active Directory objects) against the ASP.NET process identity.

- **URL Authorization**

  Configure URL Authorization in Web.config. With Forms authentication, the format of user names is determined by your custom data store; a SQL Server database, or Active Directory.

  - If you are using a SQL Server data store:

```
<authorization>
<deny users="?" />
  <allow users="Mary,Bob,Joe" roles="Manager,Sales" />
</authorization>
```

  - If you are using Active Directory as your data store, user names, and group names appear in X.500 format:

```
<authorization>
  <deny users="someAccount@domain.corp.yourCompany.com" />
  <allow roles ="CN=Smith James,CN=FTE_northamerica,CN=Users,
                DC=domain,DC=corp,DC=yourCompany,DC=com" />
</authorization>
```

Programmatic security

The following programmatic security options are available:

- **Principal Permission Demands**
  - Imperative

```
PrincipalPermission permCheck = new PrincipalPermission(
                                        null, "Manager");
permCheck.Demand();
```

  - Declarative

```
[PrincipalPermission(SecurityAction.Demand,
                    Role="Manager")]
```

- **Explicit Role Checks**. You can perform role checking using the **IPrincipal** interface.

```
IPrincipal.IsInRole("Manager");
```

When to use

Forms authentication is most ideally suited to Internet applications. Use Forms authentication when:

- Your application's users do not have Windows accounts.
- You want users to log on to your application by entering credentials using an HTML form.

More information

- For more information about Forms authentication, see "Forms Authentication" later in this chapter.
- For more information about URL authorization, see "URL Authorization Notes" later in this chapter.

Passport Authentication

The following configuration elements show how you enable Passport authentication declaratively in Web.config.

```
<authentication mode="Passport" />
```

When to use

Passport authentication is used on the Internet when application users do not have Windows accounts and you want to implement a single-sign-on solution. Users who have previously logged on with a Passport account at a participating Passport site will not have to log on to your site configured with Passport authentication.

Configuring Security

This section shows you the practical steps required to configure security for an ASP.NET Web application. These are summarized in Figure 8.3.

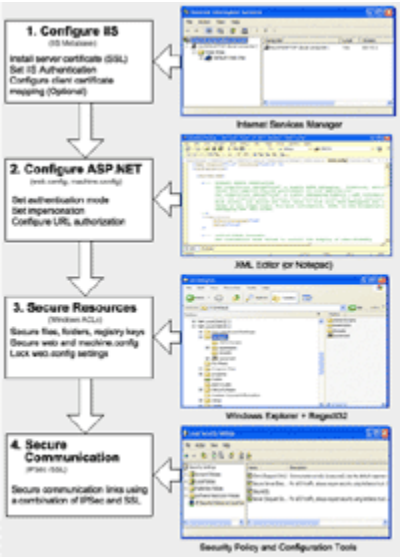

Figure 8.3. Configuring ASP.NET application security (click thumbnail for larger image)

Configure IIS Settings

To configure IIS security, you must perform the following steps:

1. Optionally install a Web server certificate (if you need SSL).

   For more information, see How To: Set Up SSL on a Web Server in the Reference section of this guide.

2. Configure IIS authentication.
3. Optionally configure client certificate mapping (if using certificate authentication).

For more information about client certificate mapping, see article Q313070, How to Configure Client Certificate Mappings in Internet Information Services (IIS) 5.0 in the Microsoft Knowledge Base.

4. Set NTFS permissions on files and folders. Between them, IIS and the ASP.NET **FileAuthorizationModule** check that the authenticated user (or the anonymous Internet user account) has the necessary access rights (based on ACL settings) to access the requested file.

Configure ASP.NET Settings

Application level configuration settings are maintained in Web.config files, which are located in your application's virtual root directory and optionally within additional subfolders (these settings can sometimes override the parent folder settings).

1. **Configure authentication**. This should be set on a per-application basis (not in Machine.config) in the Web.config file located in the application's virtual root directory.

```
2.      <authentication mode="Windows|Forms|Passport|None" />
```

3. **Configure Impersonation**. By default, ASP.NET applications do not impersonate. The applications run using the configured ASP.NET process identity (usually ASPNET) and all resource access performed by your application uses this identity. You only need impersonation in the following circumstances:

   - You are using Enterprise Services and you want to use Enterprise Services (COM+) roles to authorize access to functionality provided by serviced components.
   - IIS is configured for Anonymous authentication and you want to use the anonymous Internet user account for resource access. For details about this approach, see "Accessing Network Resources" later in this chapter.
   - You need to flow the authenticated user's security context to the next tier (for example, the database).
   - You have ported a classic ASP application to ASP.NET and want the same impersonation behavior. Classic ASP impersonates the caller by default.

   To configure ASP.NET impersonation use the following <**identity**> element in your application's Web.config.

```
<identity impersonate="true" />
```

4. **Configure Authorization**. URL authorization determines whether a user or role can issue specific HTTP verbs (for example, GET, HEAD, and POST) to a specific file. To implement URL authorization, you perform the following tasks.

   a. Add an <**authorization**> element to the Web.config file located in your application's virtual root directory.

   b. Restrict access to users and roles by using **allow** and **deny** attributes. The following example from Web.config uses Windows authentication and allows Bob and Mary access but denies everyone else.

```
c.   <authorization>

d.     <allow users="DomainName\Bob, DomainName\Mary" />

e.     <deny users="*" />

f.   </authorization>
```

> **Important** You need to add either <deny users="?"/> or <deny users="*"/> at the end of the <authorization> element, otherwise access is granted to all authenticated identities.

URL authorization notes

Take note of the following when you configure URL authorization:

- "*" refers to all identities.
- "?" refers to unauthenticated identities (that is, the anonymous identity).
- You don't need to impersonate for URL authorization to work.
- Authorization settings in Web.config usually refer to all of the files in the current directory and all subdirectories (unless a subdirectory contains its own Web.config with an <**authorization**> element. In this case the settings in the subdirectory override the parent directory settings).

  Note URL authorization only applies to file types that are mapped by IIS to the ASP.NET ISAPI extension, aspnet_isapi.dll.

  You can use the <**location**> tag to apply authorization settings to an individual file or directory. The following example shows how you can apply authorization to a specific file (Page.aspx).

```
<location path="page.aspx" />

  <authorization>

    <allow users="DomainName\Bob, DomainName\Mary" />

    <deny users="*" />

  </authorization>

</location>
```

- Users and roles for URL authorization are determined by your authentication settings:
  - When you have <**authentication mode**="Windows" /> you are authorizing access to Windows user and group accounts.

    User names take the form "DomainName\WindowsUserName"

    Role names take the form "DomainName\WindowsGroupName"

    Note The local administrators group is referred to as "BUILTIN\Administrators". The local users group is referred to as "BUILTIN\Users".

  - When you have <**authentication mode**="Forms" /> you are authorizing against the user and roles for the **IPrincipal** object that was stored in the current HTTP context. For example, if you used Forms to authenticate users against a database, you will be authorizing against the roles retrieved from the database.
  - When you have <**authentication mode**="Passport" /> you authorize against the Passport User ID (PUID) or roles retrieved from a store. For example, you can map a PUID to a particular account and set of roles stored in a SQL Server database or Active Directory.

    Note This functionality will be built into the Microsoft Windows Server 2003 operating system.

  - When you have <**authentication mode**="None" /> you may not be performing authorization. "None" specifies that you don't want to perform any authentication or that you don't want to use any of the .NET authentication modules and want to use your own custom mechanism.

    However, if you use custom authentication, you should create an **IPrincipal** object with roles and store it into the **HttpContext.User**. When you subsequently perform URL authorization, it is performed against the user and roles (no matter how they were retrieved) maintained in the **IPrincipal** object.

URL authorization examples

The following list shows the syntax for some typical URL authorization examples:

- Deny access to the anonymous account

```
<deny users="?" />
```

- Deny access to all users

```
<deny users="*"/>
```

- Deny access to Manager role

```
<deny roles="Manager"/>
```

- Forms authentication example

```
<configuration>
  <system.web>
      <authentication mode="Forms">
        <forms name=".ASPXUSERDEMO"
               loginUrl="login.aspx"
               protection="All" timeout="60" />
      </authentication>
      <authorization>
        <deny users="jdoe@somewhere.com" />
        <deny users="?" />
      </authorization>
  </system.web>
</configuration>
```

More information

The <**authorization**> element works against the current **IPrincipal** object stored in **HttpContext.User** and also the **HttpContext.Request.RequestType**.

Secure Resources

Use Windows ACLs to secure resources that include files, folders, and registry keys.

If you are not impersonating, any resource your application is required to access must have an ACL that grants at least read access to the ASP.NET process account.

If you are impersonating, files and registry keys must have an ACL that grants at least read access to the authenticated user (or the anonymous Internet user account, if anonymous authentication is in effect).

Secure Web.config and Machine.config:

- **Use the Correct ACLs**. If ASP.NET is impersonating, the impersonated identity requires read access. Otherwise, the ASP.NET process identity requires read access. Use the following ACL on Web.config and Machine.config.

  System: Full Control

  Administrators: Full Control

  Process Identity or Impersonated Identity : Read

  If you are not impersonating the anonymous Internet user account (IUSR_MACHINE), you should deny access to this account.

  > **Note** If your application is mapped to a UNC share then the UNC identity requires read access to the configuration files as well.

- Remove Unwanted HTTP Modules. Machine.config contains a set of default HTTP modules (within the <**httpModules**> element. These include:

  - WindowsAuthenticationModule
  - FormsAuthenticationModule
  - PassportAuthenticationModule
  - UrlAuthorizationModule
  - FileAuthorizationModule
  - OutputCacheModule
  - SessionStateModule

  If your application doesn't use a specific module, remove it to prevent any potential future security issues associated with a particular module from being exploited within your application.

Optionally, lock configuration settings by using the <**location**> element together with the **allowOverride="false"** attribute as described below.

Locking configuration settings

Configuration settings are hierarchical. Web.config file settings in subdirectories override Web.config settings in parent directories. Also, Web.config settings override Machine.config settings.

You can lock configuration settings to prevent them being overridden at lower levels, by using the <**location**> element coupled with the **allowOverride** attribute. For example:

```
<location path="somepath" allowOverride="false" />

 . . . arbitrary configuration settings . . .

</location>
```

Note that the path may refer to a Web site or virtual directory and it applies to the nominated directory and all subdirectories. If you set **allowOverride** to false, you prevent any lower level configuration file from overriding the settings specified in the <**location**> element. The ability to lock down configuration settings applies to all types of setting and not just security settings such as authentication modes.

Preventing files from being downloaded

You can use the **HttpForbiddenHandler** class to prevent certain file types from being downloaded over the Web. This class is used internally by ASP.NET to prevent the download of certain system level files (for example, configuration files including web.config). For a complete list of file types restricted in this way, see the <**httpHandlers**> section in machine.config.

You should consider using the **HttpForbiddenHandler** for files that your application uses internally, but are not intended for download.

Note You must also secure the files with Windows ACLs to control which users can access the files, when logged on to the Web server.

To use the HttpForbiddenHandler to prevent a particular file type from being downloaded

1. Create an application mapping in IIS for the specified file type to map it to Aspnet_isapi.dll.
   a. On the taskbar, click the **Start** button, click **Programs**, click **Administrative Tools**, and then select Internet Information Services.
   b. Select your application's virtual directory, right-click, and then click **Properties**.
   c. Select **Application Settings**, click **Configuration**..
   d. Click **Add** to create a new application mapping.
   e. Click **Browse,** and select c:\winnt\Microsoft.NET\Framework\v1.0.3705\aspnet_isapi.dll.
   f. Enter the file extension for the file type you want to prevent being downloaded (for example, .abc) in the **Extension** field.
   g. Ensure **All Verbs** and **Script engine** is selected and **Check that file exists** is not selected.
   h. Click **OK** to close the **Add/Edit Application Extension Mapping** dialog box.
   i. Click **OK** to close the **Application Configuration** dialog box, and then click **OK** again to close the **Properties** dialog box.
2. Add an <**HttpHandler**> mapping in Web.config for the specified file type.

   An example for the .abc file type is shown below.

```
<httpHandlers>
  <add verb="*" path="*.abc"
    type="System.Web.HttpForbiddenHandler"/>
</httpHandlers>
```

Secure Communication

Use a combination of SSL and Internet Protocol Security (IPSec) to secure communication links.

More information

- For information about using SSL to secure the link to the database server, see How To: Use SSL to Secure Communication with SQL Server 2000.
- For information about using IPSec between two computers, see How To: Use IPSec to Provide Secure Communication Between Two Servers.

Programming Security

After you establish your Web application's configurable security settings, you need to further enhance and fine-tune your application's authorization policy programmatically. This includes using declarative .NET attributes within your assemblies and performing imperative authorizing checks within code.

This section highlights the key programming steps required to perform authorization within an ASP.NET Web application.

An Authorization Pattern

The following summarizes the basic pattern for authorizing users within your Web application:

1. Retrieve credentials
2. Validate credentials
3. Put users in roles
4. Create an **IPrincipal** object
5. Put the **IPrincipal** object into the current HTTP context
6. Authorize based on the user identity / role membership

Important Steps 1 to 5 are performed automatically by ASP.NET if you have configured Windows authentication. For other authentication mechanisms (Forms, Passport and custom approaches), you must write code to perform these steps, as discussed below.

Retrieve credentials

You must start by retrieving a set of credentials (user name and password) from the user. If your application does not use Windows authentication, you need to ensure that clear text credentials are properly secured on the network by using SSL.

Validate credentials

If you have configured Windows authentication, credentials are validated automatically using the underlying services of the operating system.

If you use an alternate authentication mechanism, you must write code to validate credentials against a data store such as a SQL Server database or Active Directory.

For more information about how to securely store user credentials in a SQL Server database, see Authenticating Users Against a Database within Chapter 12, "Data Access Security."

Put users in roles

Your user data store should also contain a list of roles for each user. You must write code to retrieve the role list for the validated user.

Create an IPrincipal object

Authorization occurs against the authenticated user, whose identity and role list is maintained within an **IPrincipal** object (which flows in the context of the current Web request).

If you have configured Windows authentication, ASP.NET automatically constructs a **WindowsPrincipal** object. This contains the authenticated user's identity together with a role list, which equates to the list of Windows groups to which the user belongs.

If you are using Forms, Passport, or custom authentication, you must write code within the **Application_AuthenticateRequest** event handler in Global.asax to create an **IPrincipal** object. The **GenericPrincipal** class is provided by the .NET Framework, and should be used in most scenarios.

Put the IPrincipal object into the current HTTP context

Attach the **IPrincipal** object to the current HTTP context (using the Http**Context.User** variable). ASP.NET does this automatically when you use Windows authentication. Otherwise, you must attach the object manually.

Authorize based on the user identity and/or role membership

Use .NET roles either declaratively (to obtain class or method level authorization), or imperatively within code if your application requires more fine-grained authorization logic.

You can use declarative or imperative principal permission demands (using the **PrincipalPermission** class), or you can perform explicit role checks by calling the **IPrincipal.IsInRole()** method.

The following example assumes Windows authentication and shows a declarative principal permission demand. The method that follows the attribute will only be executed if the authenticated user is a member of the **Manager** Windows group. If the caller is not a member of this group, a **SecurityException** is thrown.

```
 [PrincipalPermission(SecurityAction.Demand,

                      Role=@"DomainName\Manager")]

public void SomeMethod()

{

}
```

The following example shows an explicit role check within code. This example assumes Windows authentication. If a non-Windows authentication mechanism is used, the code remains very similar. Instead of casting the **User** object to a **WindowsPrincipal** object, it should be cast to a **GenericPrincipal** object.

```
// Extract the authenticated user from the current HTTP context.

// The User variable is equivalent to HttpContext.Current.User if you

  are

using // an .aspx or .asmx page

WindowsPrincipal authenticatedUser = User as WindowsPrincipal;

if (null != authenticatedUser)

{

  // Note: To retrieve the authenticated user's username, use the

  // following line of code

  // string username = authenticatedUser.Identity.Name;



  // Perform a role check

  if (authenticatedUser.IsInRole(@"DomainName\Manager") )

  {
```

```
    // User is authorized to perform manager functionality
  }
}
else
{
  // User is not authorized to perform manager functionality
}
```

More information

- For a practical implementation of the above pattern for Forms authentication, see the "Forms Authentication" section later in this chapter.

Creating a Custom IPrincipal class

The **GenericPrincipal** class provided by the .NET Framework should be used in most circumstances when you are using a non-Windows authentication mechanism. This provides role checks using the **IPrincipal.IsInRole** method.

On occasion, you may need to implement your own **IPrincipal** class. Reasons for implementing your own **IPrincipal** class include:

- You want extended role checking functionality. You might want methods that allow you to check whether a particular user is a member of multiple roles. For example:

```
CustomPrincipal.IsInAllRoles( "Role", "Role2", "Role3" )
CustomPrincipal.IsInAnyRole( "Role1", "Role2", "Role3" )
```

- You may want to implement an extra method or property that returns a list of roles in an array. For example:

```
string[] roles = CustomPrincipal.Roles;
```

- You want your application to enforce role hierarchy logic. For example, a Senior Manager may be considered higher up in the hierarchy than a Manager. This could be tested using methods like the ones shown below.

```
CustomPrincipal.IsInHigherRole("Manager");
CustomPrincipal.IsInLowerRole("Manager");
```

- You may want to implement lazy initialization of the role lists. For example, you could dynamically load the role list only when a role check is requested.
- You may want to implement the **IIdentity** interface to have the user identified by an **X509ClientCertificate**. For example:

```
CustomIdentity id = CustomPrincipal.Identity;
X509ClientCertificate cert = id.ClientCertificate;
```

More information

For more information about creating your own **IPrincipal** class, see How To: Implement IPrincipal in the Reference section of this guide.

Windows Authentication

Use Windows authentication when the users of your application have Windows accounts that can be authenticated by the server (for example, in intranet scenarios).

If you configure ASP.NET for Windows authentication, IIS performs user authentication by using the configured IIS authentication mechanism. This is shown in Figure 8.4.

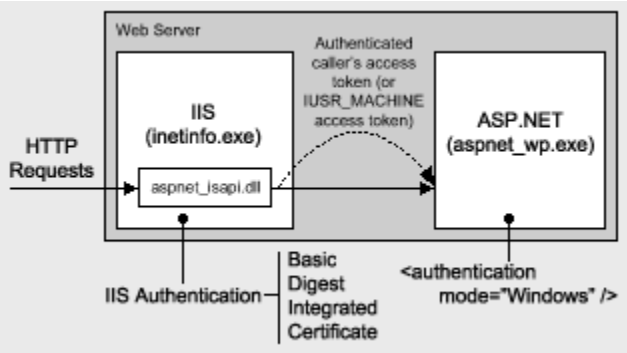

Figure 8.4. ASP.NET Windows authentication uses IIS to authenticate callers

The access token of the authenticated caller (which may be the Anonymous Internet user account if IIS is configured for Anonymous authentication) is made available to the ASP.NET application. Note the following:

- This allows the ASP.NET **FileAuthorizationModule** to perform access checks against requested ASP.NET files using the original caller's access token.

  Important ASP.NET File authorization only performs access checks against file types that are mapped to Aspnet_isapi.dll.

- File authorization does not require impersonation. With impersonation enabled any resource access performed by your application uses the impersonated caller's identity. In this event, ensure that the ACLs attached to resources contain an Access Control Entry (ACE) that grants at least read access to the original caller's identity.

Identifying the authenticated user

ASP.NET associates a **WindowsPrincipal** object with the current Web request. This contains the identity of the authenticated Windows user together with a list of roles that the user belongs to. With Windows authentication, the role list consists of the set of Windows groups to which the user belongs.

The following code shows how to obtain the identity of the authenticated Windows user and to perform a simple role test for authorization.

```
WindowsPrincipal user = User as WindowsPrincipal;

if (null != user)

{

  string username = user.Identity.Name;

  // Perform a role check
```

```
  if ( user.IsInRole(@"DomainName\Manager") )
  {
    // User is authorized to perform manager functionality
  }
}
else
{
  // Throw security exception as we don't have a WindowsPrincipal
}
```

Forms Authentication

When you are using Forms authentication, the sequence of events triggered by an unauthenticated user who attempts to access a secured file or resource (where URL authorization denies the user access), is shown in Figure 8.5.

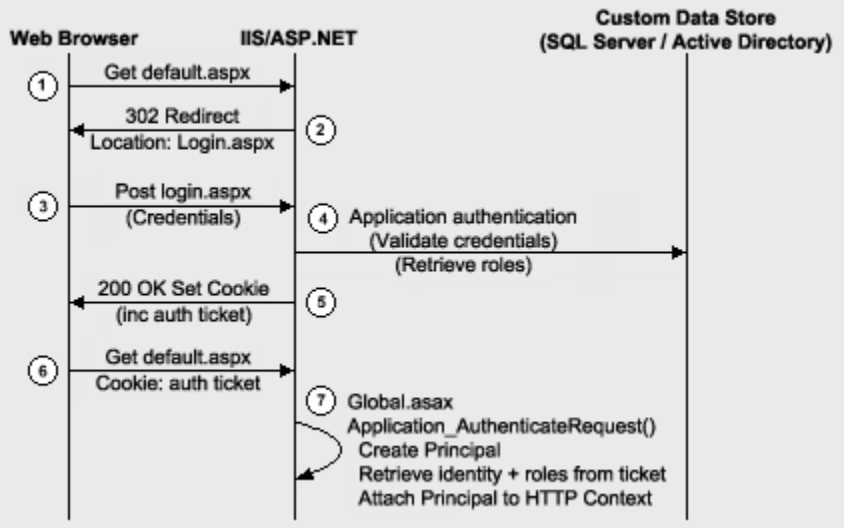

Figure 8.5. Forms authentication sequence of events

The following describes the sequence of events shown in Figure 8.5:

1. The user issues a Web request for Default.aspx.

   IIS allows the request because Anonymous access is enabled. ASP.NET checks the <**authorization**> elements and finds a <**deny users**=?" /> element.

2. The user is redirected to the login page (Login.aspx) as specified by the **LoginUrl** attribute of the <**forms**> element.

3. The user supplies credentials and submits the login form.

4. The credentials are validated against a store (SQL Server or Active Directory) and roles are optionally retrieved. You must retrieve a role list if you want to use role-based authorization.

5. A cookie is created with a **FormsAuthenticationTicket** and sent back to the client. Roles are optionally stored in the ticket. By storing the role list in the ticket, you avoid having to access the database to re-retrieve the list for each successive Web request from the same user.

6. The user is redirected with client-side redirection to the originally requested page (Default.aspx).

7. In the **Application_AuthenticateRequest** event handler (in Global.asax), the ticket is used to create an **IPrincipal** object and it is stored in **HttpContext.User**.

   ASP.NET checks the <**authorization**> elements and finds a <**deny users**=?" /> element. However, this time the user is authenticated.

   ASP.NET checks the <**authorization**> elements to ensure the user is in the <**allow**> element.

   The user is granted access to Default.aspx.

Development Steps for Forms Authentication

The following list highlights the key steps that you must perform to implement Forms authentication:

1. Configure IIS for anonymous access.
2. Configure ASP.NET for Forms authentication.
3. Create a logon Web form and validate the supplied credentials.
4. Retrieve a role list from the custom data store.
5. Create a Forms authentication ticket (store roles in the ticket).
6. Create an **IPrincipal** object.
7. Put the **IPrincipal** object into the current HTTP context.
8. Authorize the user based on user name/role membership.

Configure IIS for anonymous access

Your application's virtual directory must be configured in IIS for anonymous access.

To configure IIS for anonymous access

1. Start the Internet Information Services administration tool.
2. Select your application's virtual directory, right-click, and then click **Properties**.
3. Click **Directory Security**.
4. In the **Anonymous access and authentication control** group, click **Edit**.
5. Select **Anonymous access**.

Configure ASP.NET for Forms authentication

A sample configuration is shown below.

```
<authentication mode="Forms">
  <forms name="MyAppFormsAuth"
       loginUrl="login.aspx"
       protection="Encryption"
```

```
        timeout="20"
        path="/" >
  </forms>
</authentication>
```

Create a logon Web form and validate the supplied credentials

Validate credentials against a SQL Server database, or Active Directory.

More information

- See "How To: Use Forms Authentication with SQL Server 2000" in the Reference section of this guide.
- See "How To: Use Forms Authentication with Active Directory" in the Reference section of this guide.

Retrieve a role list from the custom data store

Obtain roles from a table within a SQL Server database, or groups/distribution lists configured within Active Directory. Refer to the preceding resources for details.

Create a Forms authentication ticket

Store the retrieved roles in the ticket. This is illustrated in the following code.

```
// This event handler executes when the user clicks the Logon button
// having supplied a set of credentials
private void Logon_Click(object sender, System.EventArgs e)
{
  // Validate credentials against either a SQL Server database
  // or Active Directory
  bool isAuthenticated = IsAuthenticated( txtUserName.Text,
                                          txtPassword.Text );
  if (isAuthenticated == true )
  {
    // Retrieve the set of roles for this user from the SQL Server
    // database or Active Directory. The roles are returned as a
    // string that contains pipe separated role names
    // for example "Manager|Employee|Sales|"
    // This makes it easy to store them in the authentication ticket
```

```
string roles = RetrieveRoles( txtUserName.Text, txtPassword.Text
  );

// Create the authentication ticket and store the roles in the
// custom UserData property of the authentication ticket
FormsAuthenticationTicket authTicket = new
   FormsAuthenticationTicket(
               1,                          // version
               txtUserName.Text,           // user name
               DateTime.Now,               // creation
               DateTime.Now.AddMinutes(20),// Expiration
               false,                      // Persistent
               roles );                    // User data
 // Encrypt the ticket.
 string encryptedTicket =
   FormsAuthentication.Encrypt(authTicket);
 // Create a cookie and add the encrypted ticket to the
 // cookie as data.
 HttpCookie authCookie =
           new HttpCookie(FormsAuthentication.FormsCookieName,
                          encryptedTicket);

 // Add the cookie to the outgoing cookies collection.
 Response.Cookies.Add(authCookie);
 // Redirect the user to the originally requested page
 Response.Redirect( FormsAuthentication.GetRedirectUrl(
                    txtUserName.Text,
```

```
                          false ));
  }
}
```

Create an IPrincipal object

Create the **IPrincipal** object in the **Application_AuthenticationRequest** event handler in Global.asax. Use the **GenericPrincipal** class, unless you need extended role-based functionality. In this case create a custom class that implements **IPrincipal**.

Put the IPrincipal object into the current HTTP context

The creation of a **GenericPrincipal** object is shown below.

```
protected void Application_AuthenticateRequest(Object sender,
  EventArgs e)
{
  // Extract the forms authentication cookie
  string cookieName = FormsAuthentication.FormsCookieName;
  HttpCookie authCookie = Context.Request.Cookies[cookieName];
  if(null == authCookie)
  {
    // There is no authentication cookie.
    return;
  }
  FormsAuthenticationTicket authTicket = null;
  try
  {
    authTicket = FormsAuthentication.Decrypt(authCookie.Value);
  }
  catch(Exception ex)
  {
    // Log exception details (omitted for simplicity)
    return;
  }
```

```
  if (null == authTicket)
  {
    // Cookie failed to decrypt.
    return;
  }
  // When the ticket was created, the UserData property was assigned
  // a pipe delimited string of role names.
  string[] roles = authTicket.UserData.Split(new char[]{'|'});

  // Create an Identity object
  FormsIdentity id = new FormsIdentity( authTicket );
  // This principal will flow throughout the request.
  GenericPrincipal principal = new GenericPrincipal(id, roles);
  // Attach the new principal object to the current HttpContext
    object
  Context.User = principal;
}
```

Authorize the user based on user name or role membership

Use declarative principal permission demands to restrict access to methods. Use imperative principal permission demands and/or explicit role checks (**IPrincipal.IsInRole**) to perform fine-grained authorization within methods.

Forms Implementation Guidelines

- Use SSL when capturing credentials using an HTML form.

  In addition to using SSL for the login page, you should also use SSL for other pages, whenever the credentials or the authentication cookie is sent across the network. This is to mitigate the threat associated with cookie replay attacks.

- Authenticate users against a custom data store. Use SQL Server or Active Directory.
- Retrieve a role list from the custom data store and store a delimited list of roles within the **UserData** property of the **FormsAuthenticationTicket** class. This improves performance by eliminating repeated access to the data store for each Web request and also saves you from storing the user's credentials in the authentication cookie.
- If the list of roles is extensive and there is a danger of exceeding the cookie size limit, store the role details in the ASP.NET cache object or database and retrieve them on each subsequent request.
- For each request after initial authentication:

- Retrieve the roles from the ticket in the **Application_AuthenticateRequest** event handler.
- Create an **IPrincipal** object and store it in the HTTP context (**HttpContext.User**). The .NET Framework also associates it with the current .NET thread (**Thread.CurrentPrincipal**).
- Use the **GenericPrincipal** class unless you have a specific need to create a custom **IPrincipal** implementation; for example, to support enhanced role-based operations.

- Use two cookies; one for personalization and one for secure authentication and authorization. Make the personalization cookie persistent (make sure it does not contain information that would permit a request to perform a restricted operation; for example, placing an order within a secure part of a site).
- Use a separate cookie name (using the **Forms** attribute of the <**forms**> element) and path for each Web application. This will ensure that users who are authenticated against one application are not treated as authenticated when using a second application hosted by the same Web server.
- Ensure cookies are enabled within client browsers. For a Forms authentication approach that does not require cookies, see "Cookieless Forms Authentication" later in this chapter.

More information

- See How To: Use Forms Authentication with SQL Server 2000 in the Reference section of this guide.
- See How To: Use Forms Authentication with Active Directory in the Reference section of this guide.
- See How To: Create GenericPrincipal Objects with Forms Authentication in the Reference section of this guide.

Hosting Multiple Applications Using Forms Authentication

If you are hosting multiple Web applications that use Forms authentication on the same Web server, it is possible for a user who is authenticated in one application to make a request to another application without being redirected to that application's logon page. The URL authorization rules within the second application may deny access to the user, without providing the opportunity to supply logon credentials using the logon form.

This only happens if the name and path attributes on the <**forms**> element are the same across multiple applications and each application uses a common <**machineKey**> element in Web.config.

More information

For more information about this issue, and for resolution techniques, see the following Knowledge Base articles:

- Q313116, PRB: Forms Authentication Requests Are Not Directed to loginUrl Page
- Q310415, PRB: Mobile Forms Authentication and Different Web Applications

Cookieless Forms Authentication

If you need a cookieless Forms authentication solution, consider using the approach used by the Microsoft Mobile Internet Toolkit. Mobile Forms Authentication builds upon Forms Authentication but uses the query string to convey the authentication ticket instead of a cookie.

More information

For more information about Mobile Forms Authentication, see article Q311568, INFO: How To Use Mobile Forms Authentication with Microsoft Mobile Internet Toolkit, in the Microsoft Knowledge Base.

Passport Authentication

Use Passport authentication when the users of your application have Passport accounts and you want to implement a single-sign-on solution with other Passport enabled sites.

When you configure ASP.NET for Passport authentication, the user is prompted to log in and then is redirected to the Passport site. After successful credential validation, the user is redirected back to your site.

Configure ASP.NET for Passport authentication

To configure ASP.NET for Passport authentication, use the following Web.config settings.

```
<authentication mode="Passport">
  <passport redirectUrl="internal" />
</authentication>
<authorization>
  <deny users="?" />
  <allow users="*" />
</authorization>
```

Map a Passport identity into roles in Global.asax

To map a Passport identity into roles, implement the **PassportAuthentication_OnAuthentication** event handler in Global.asax as shown below.

```
void PassportAuthentication_OnAuthenticate(Object sender,
                                   PassportAuthenticationEventArgs e)
{
  if(e.Identity.Name == "0000000000000001")
  {
    string[] roles = new String[]{"Developer", "Admin", "Tester"};
    Context.User = new GenericPrincipal(e.Identity, roles);
  }
}
```

Test role membership

The following code fragment shows how to retrieve the authenticated Passport identity and check role membership within an aspx page.

```
PassportIdentity passportId = Context.User.Identity as
  PassportIdentity;
if (null == passportId)
{
  Response.Write("Not a PassportIdentity<br>");
  return;
```

```
}

Response.Write("IsInRole: Develeoper? " +

  Context.User.IsInRole("Developer"));
```

Custom Authentication

If none of the authentication modules supplied with the .NET Framework meet your precise authentication needs, you can use custom authentication and implement your own authentication mechanism. For example, your company may already have a custom authentication strategy that is widely used by other applications.

To implement custom authentication in ASP.NET:

- Configure the authentication mode in Web.config as shown below. This notifies ASP.NET that it should not invoke any of its built-in authentication modules.

```
<authentication mode="None" />
```

- Create a class that implements the **System.Web.IHttpModule** interface to create a custom HTTP module. This module should hook into the **HttpApplication.AuthenticateRequest** event and provide a delegate to be called on each request to the application when authentication is required.

  An authentication module must:

  - Obtain credentials from the caller.
  - Validate the credentials against a store.
  - Create an **IPrincipal** object and store it in **HttpContext.User.**
  - Create and protect an authentication token and send it back to the user (typically in a query string, cookie, or hidden form field).
  - Obtain the authentication token on subsequent requests, validate it, and reissue it.

More information

For more information about how to implement a custom HTTP Module, see article Q307996, HOW TO: Create an ASP.NET HTTP Module Using Visual C# .NET, in the Microsoft Knowledge Base.

Process Identity for ASP.NET

Run ASP.NET (specifically the Aspnet_wp.exe worker process) by using a least privileged account.

Use a Least Privileged Account

Use a least privileged account to lessen the threat associated with a process compromise. If a determined attacker manages to compromise the ASP.NET process that runs your Web application, they can easily inherit and exploit the privileges and access rights granted to the process account. An account configured with minimum privileges restricts the potential damage that can be done.

Avoid Running as SYSTEM

Don't use the highly-privileged SYSTEM account to run ASP.NET and don't grant the ASP.NET process account the "Act as part of the operating system" privilege. You may be tempted to do one or the other to allow your code to call the **LogonUser** API to obtain a fixed identity (typically for network resource access). For alternate approaches, see "Accessing Network Resources" later in this chapter.

Reasons for not running as SYSTEM, or granting the "Act as part of the operating system privilege" include:

- It significantly increases the damage that an attacker can do when the system is compromised, but it doesn't affect the ability to be compromised.
- It defeats the principle of least privilege. The ASPNET account has been specifically configured as a least privileged account designed to run ASP.NET Web applications.

More information

For more information about the "Act as part of the operating system" privilege, see the Microsoft Systems Journal August 1999 Security Briefs column.

Domain controllers and the ASP.NET process account

In general, it is not advisable to run your Web server on a domain controller, because a compromise of the server is a compromise of the domain. If you need to run ASP.NET on a domain controller, you need to give the ASP.NET process account appropriate privileges as outlined in article Q315158, BUG: ASP.NET Does Not Work with the Default ASPNET Account on a Domain Controller, in the Microsoft Knowledge Base.

Using the Default ASPNET Account

The local ASPNET account has been configured specifically to run ASP.NET Web applications with the minimum possible set of privileges. Use ASPNET whenever possible.

By default, ASP.NET Web applications run using this account, as configured by the <**processModel**> element within Machine.config.

```
<processModel userName="machine" password="AutoGenerate" />
```

Note The *machine* user name indicates the ASPNET account. The account is created with a cryptographically strong password when you install the .NET Framework. In addition to being configured within the Security Account Manager (SAM) database, the password is stored within the Local System Authority (LSA) on the local computer. The system retrieves the password from the LSA, when it launches the ASP.NET worker process.

If your application accesses network resources, the ASPNET account must be capable of being authenticated by the remote computer. You have two choices:

- Reset the ASPNET account's password to a known value and then create a duplicate account (with the same name and password) on the remote computer. This approach is the only option in the following circumstances:
  - The Web server and remote computer are in separate domains with no trust relationship.
  - The Web server and remote computer are separated by a firewall and you do not want to open the necessary ports to support Windows authentication.
- If ease of administration is your primary concern, use a least privileged, domain account.

  To avoid having to manually update and synchronize passwords, you can use a least privileged domain account to run ASP.NET. It is vital that the domain account is fully locked down to mitigate the process compromise threat. If an attacker manages to compromise the ASP.NET worker process, he or she will have the ability to access domain resources, unless the account is fully locked down.

  > **Note** If you use a local account and the account becomes compromised, the only computers subject to attack are the computers on which you have created duplicate accounts. If you use a domain account, the account is visible to each computer on the domain. However, the account still needs to have permission to access those computers.

The <processModel> element

The <**processModel**> element in the Machine.config file contains the **userName** and **password** attributes which specify the account that should be used to run the ASP.NET worker process (Aspnet_wp.exe). You have a number of options for configuring this setting. For example:

- **"machine"**. The worker process runs as the default least privileged ASPNET account. The account has network access but cannot be authenticated to any other computer on the network because the account is local to the computer and there is no authority to vouch for the account. On the network, this account is represented as "MachineName\ASPNET".

- **"system"**. The worker process runs as the local SYSTEM account. This account has extensive privileges on the local computer and also has the ability to access the network using the credentials of the computer. On the network, this account is represented as "DomainName\MachineName$".

- **Specific credentials**. When you supply credentials for **userName** and **password**, remember the principle of least privilege. If you specify a local account, the Web application cannot be authenticated on the network unless you create a duplicate account on the remote computer. If you elect to use a least privileged domain account, ensure it is not an account that has permission to access more computers on the network than it needs to.

  In the .NET Framework version 1.1 you will have the ability to store encrypted **userName** and **password** attributes in the registry.

Note In contrast to the way classic ASP applications run, ASP.NET code never runs in the dllhost.exe process or as the IWAM_MACHINENAME account even when the application protection level is set to **High** (**Isolated)** in IIS. ASP.NET requests sent to IIS are directly routed to the ASP.NET worker process (Aspnet_wp.exe). The ASP.NET ISAPI extension, Aspnet_isapi.dll, runs in the IIS (Inetinfo.exe) process address space. (This is controlled by the **InProcessIsapiApps** Metabase entry, which should not be modified). The ISAPI extension is responsible for routing requests to the ASP.NET worker process. ASP.NET applications then run in the ASP.NET worker process, where application domains provide isolation boundaries.
In IIS 6, you will be able to isolate ASP.NET applications by configuring application pools, where each pool will have its own application instance.

More information

- For more information about accessing network resources from ASP.NET Web applications, see "Accessing Network Resources," later in this chapter.

- For detailed information about how to create a custom account for running ASP.NET, see How To: Create a Custom Account to Run ASP.NET in the Reference section of this guide.

Impersonation

With the introduction of the **FileAuthorizationModule**, and with the efficient use of gatekeepers and trust boundaries, impersonation may prove more of a disadvantage than a benefit in ASP.NET.

Impersonation and Local Resources

If you use impersonation and access local resources from your Web application code, you must configure the ACLs attached to each secured resource to contain an ACE that grants at least read access to the authenticated user.

A better approach is to avoid impersonation, grant permissions to the ASP.NET process account, and use URL authorization, File authorization, and a combination of declarative and imperative role-based checks.

Impersonation and Remote Resources

If you use impersonation and then access remote resources from your Web application code, the access will fail unless you are using Basic, Forms, or Kerberos authentication. If you use Kerberos authentication, user accounts must be suitably configured for delegation. They must be marked as " Sensitive and cannot be delegated" within Active Directory.

Note: Windows 2000 SP 4 introduces a new user right in the security policy called "Impersonate a client after authentication". You may need this right when when you are impersonating access remote resources and shares. When using impersonation under ASP.NET and accessing remote resources and shares, you may see the following errors

- An error occurred while trying to load the string resources (getmodulehandle) failed with error -2147023888

- "Server application unavailable" and the event logs would show a 1000 error for ASP.NET.
- System.Web.HttpException: An error occurred while try to load the string resources (GetModuleHandle failed with error 126)
- Exception Details: System.ApplicationException: Access is denied.
- COM object with CLSID {A48ECD2F-169C-4F1A-BFC7-650D38BAB4F4} is either not valid or not registered.

More information

For more information about how to configure Kerberos delegation, see:

- Flowing the Original Caller to the Database in Chapter 5, "Intranet Security."
- "How To: Implement Kerberos Delegation for Windows 2000" in the Reference section of this guide.

Impersonation and Threading

If a thread that is impersonating creates a new thread, the new thread inherits the security context of the ASP.NET process account and not the impersonated account.

Accessing System Resources

ASP.NET performs no impersonation by default. As a result, if your Web application accesses local system resources, it does so using the security context associated with the Aspnet_wp.exe worker process. The security context is determined by the account used to run the worker process.

Accessing the Event Log

Least privileged accounts have sufficient permissions to be able to write records to the event log by using existing event sources. However, they do not have sufficient permissions to create new event sources. This requires a new entry to be placed beneath the following registry hive.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\<log>
```

To avoid this issue, create the event sources used by your application at installation time, when administrator privileges are available. A good approach is to use a .NET installer class, which can be instantiated by the Windows Installer (if you are using .msi deployment) or by the InstallUtil.exe system utility if you are not.

If you are unable to create event sources at installation time, you must add permission to the following registry key and grant access to the ASP.NET process account (of any impersonated account if your application uses impersonation).

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog
```

The account(s) must have the following minimum permissions:

- Query key value
- Set key value
- Create subkey
- Enumerate subkeys
- Notify
- Read

The following code can be used to write to the Application event log from ASP.NET once permissions have been applied to the registry:

```
string source = "Your Application Source";

string logToWriteTo = "Application";

string eventText = "Sample Event";


if (!EventLog.SourceExists(source))

{

  EventLog.CreateEventSource(source, logToWriteTo);

}

EventLog.WriteEntry(source, eventText, EventLogEntryType.Warning,

  234);
```

Accessing the Registry

Any registry key accessed by your application requires an ACE in the ACL that grants (at minimum) read access to the ASP.NET process account.

More information

For more information about installer classes and the InstallUtil.exe utility, see the .NET Framework Tools on MSDN.

Accessing COM Objects

In classic ASP, requests are processed using threads from the Single Threaded Apartment (STA) thread pool. In ASP.NET, requests are processed using threads from the Multithreaded Apartment (MTA) thread pool. This has implications for ASP.NET Web applications that call Apartment model objects.

Apartment Model Objects

When an ASP.NET Web application calls an Apartment model object (such as a Visual Basic 6 COM object) there are two issues to note:

- You must mark your ASP.NET page with the **AspCompat** directive, as shown below.

```
<%@ Page Language="C#" AspCompat="True" %>
```

- Don't create your COM objects outside of specific Page event handlers. Always create COM objects in Page event handlers (such as **Page_Load** and **Page_Init**). Don't create COM objects in the page's constructor.

The AspCompat directive is required

By default, ASP.NET uses MTA threads to process requests. This results in a thread-switch when an Apartment model object is called from ASP.NET, because the Apartment model object can't be accessed directly by MTA threads (COM would use an STA thread).

Specifying **AspCompat** causes the page to be processed by an STA thread. This avoids a thread switch from MTA to STA. This is important from a security perspective if your Web application is impersonating because a thread switch results in a lost impersonation token. The new thread would not have the impersonation token associated with it.

The **AspCompat** directive is not supported for ASP.NET Web services. This means that when you call Apartment model objects from Web service code, a thread switch does occur and you lose the thread impersonation token. This typically results in an Access Denied exception.

More information

- See the following Knowledge Base articles for more information:
  - Article Q303375, INFO: XML Web Services and Apartment Objects
  - Article Q325791, PRB: Access Denied Error Message Occurs When Impersonating in ASP.NET and Calling STA COM Components
- For more information about how to determine the identity of the currently executing code, see the Determining Identity section of Chapter 13, "Troubleshooting Security Issues."

Don't create COM objects outside of specific page events

Don't create COM object outside of specific Page event handlers. The following code fragment illustrates what not to do.

```
<%@ Page Language="C#" AspCompat="True" %>

<script runat="server">

  // COM object created outside of Page events

  YourComObject obj = new apartmentObject();

  public void Page_Load()

  {

    obj.Foo()

  }

</script>
```

When you use Apartment model objects, it is important to create the object within specific Page events such as **Page_Load**, as shown below.

```
<%@ Page Language="C#" AspCompat="True" %>

<script runat="server">

public void Page_Load()

{

  YourComObject obj = new apartmentObject();

  obj.Foo()

}

</script>
```

More information

For more information, see article Q308095, "PRB: Creating STA Components in the Constructor in ASP.NET ASPCOMPAT Mode Negatively Impacts Performance" in the Microsoft Knowledge Base.

C# and VB .NET objects in COM+

Microsoft C#® development tool and Microsoft Visual Basic® .NET development system support all threading models (Free-threaded, Neutral, Both, and Apartment). By default, when hosted in COM+, C# and Visual Basic .NET objects are marked as Both. As a result, when they are called by ASP.NET, access is direct and you do not incur a thread switch.

Accessing Network Resources

Your application may need to access network resources. It is important to be able to identify:

- The resources your application needs to access.

  For example, files on file shares, databases, DCOM servers, Active Directory objects, and so on.

- The identity used to perform the resource access.

  If your application accesses remote resources, this identity must be capable of being authenticated by the remote computer.

> **Note** For information specific to accessing remote SQL Server databases, see Chapter 12, Data Access Security.

You can access remote resources from an ASP.NET application by using any of the following techniques:

- Use the ASP.NET process identity.
- Use a serviced component.
- Use the Anonymous Internet user account (for example, IUSR_MACHINE).
- Use the **LogonUser** API and impersonating a specific Windows identity.
- Use the original caller.

Using the ASP.NET Process Identity

When the application is not configured for impersonation, the ASP.NET process identity provides the default identity when your application attempts to access remote resources. If you want to use the ASP.NET process account for remote resource access, you have three options:

- Use mirrored accounts.

  This is the simplest approach. You create a local account with a matching user name and password on the remote computer. You must change the ASPNET account password in User Manager to a known value (always use a strong password). You must then explicitly set this on the <**processModel**> element in Machine.config, and replace the existing "**AutoGenerate**" value.

> **Important** If you change the ASPNET password to a known value, the password in the LSA will no longer match the SAM account password. If you need to revert to the "**AutoGenerate**" default, you will need to do the following:
>
> Run Aspnet_regiis.exe, to reset ASP.NET to its default configuration. For more information, see article Q306005, HOWTO: Repair IIS Mapping After You Remove and Reinstall IIS in the Microsoft Knowledge Base.

- Create a custom, least privileged local account to run ASP.NET and create a duplicate account on the remote computer.
- Run ASP.NET using a least-privileged domain account.

  This assumes that client and server computers are in the same or trusting domains.

More information

For more information about configuring an ASP.NET process account, see "How To: Create a Custom Account to Run ASP.NET" in the Reference section of this guide.

Using a Serviced Component

You can use an out of process-serviced component, configured to run as a fixed identity to access network resources. This approach is shown in Figure 8.6.

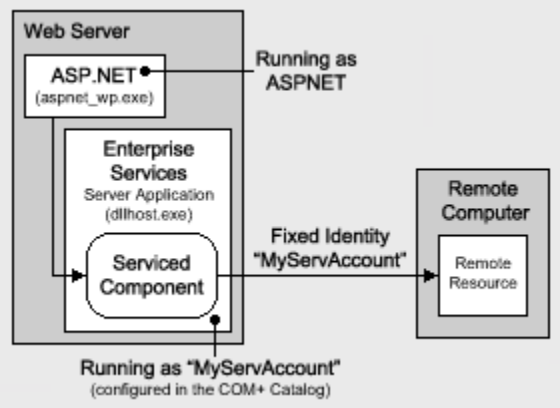

Figure 8.6. Using an out of process serviced component to provide a fixed identity for network resource access

Using an out of process-serviced component (in an Enterprise Services server application) has the following advantages:

- Flexibility in terms of the identity used. You don't just rely on the ASP.NET identity.
- Trusted or higher-privileged code can be isolated from your main Web application.
- An additional process hop raises the bar from a security perspective. It makes it much tougher for an attacker to cross the process boundary to a process with raised privileges.
- If you need to hand-craft impersonation with **LogonUser** API calls, you can do so in a process that is separated from your main Web application.

  Note To call **LogonUser** you must give the Enterprise Services process-account the "Act as part of the operating system" privilege. Raising the privileges for a process that is separate from your Web application is less of a security concern.

Using the Anonymous Internet User Account

You can use the anonymous Internet user account to access network resources if IIS is configured for Anonymous authentication. This is the case if one of the following is true:

- Your application supports anonymous access.
- Your application uses Forms, Passport, or Custom authentication (where IIS is configured for anonymous access).

To use the anonymous account for remote resource access

1. Configure IIS for Anonymous authentication. You can set the ASP.NET authentication mode to Windows, Forms, Passport, or None, depending upon the authentication requirements of your application.

2. Configure ASP.NET for impersonation. Use the following setting in Web.config:

```
3.      <identity impersonate="true" />
```

4. Configure the anonymous account as a least privileged domain account,

   —or—

   Duplicate the anonymous account by using the same user name and password on the remote computer. This approach is necessary when you are making calls across non-trusting domains or through firewalls where the necessary ports to support Integrated Windows authentication are not open.

   To support this approach, you must also:

   a. Use Internet Services Manager to clear the **Allow IIS to Control Password** checkbox for the anonymous account.

      If you select this option, the logon session created using the specified anonymous account ends up with NULL network credentials (and therefore cannot be used to access network resources). If you don't select this option, the logon session is an interactive logon session with network credentials.

   b. Set the account's credentials both in User Manager and in Internet Services Manager.

Important If you impersonate the anonymous account (for example, IUSR_MACHINE), resources must be secured against this account (using appropriately configured ACLs). Resources that your application needs to access must grant read access (at minimum) to the anonymous account. All other resources should deny access to the anonymous account.

Hosting multiple Web applications

You can use a separate anonymous Internet user account for each virtual root within your Web site. In a hosted environment, this allows you to separately authorize, track, and audit requests that originate from separate Web applications. This approach is shown in Figure 8.7.

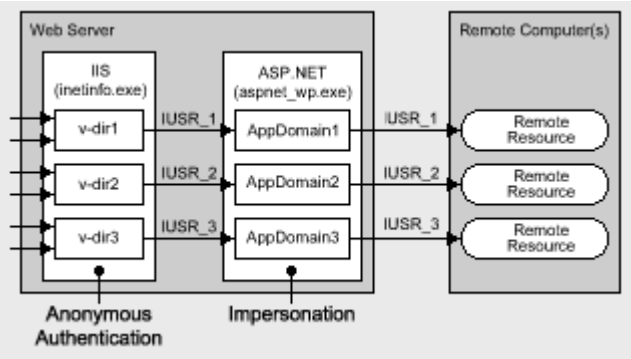

Figure 8.7. Impersonating separate anonymous Internet user accounts per application (v-dir)

To configure the anonymous Internet user account for a specific virtual directory

1. Start **Internet Services Manager** from the **Administrative Tools** programs group.
2. Select the virtual directory you want to configure, right-click, and then click **Properties**.
3. Click the **Directory Security** tab.
4. Click **Edit** within the **Anonymous access and authentication control** group.

5. Select **Anonymous access,** and then click **Edit**.

6. Enter the user name and password of the account that you want IIS to use when anonymous users connect to the site.

7. Make sure that **Allow IIS to control password** is NOT selected.

Using LogonUser and Impersonating a Specific Windows Identity

You can impersonate a specific identity by configuring user name and password attributes on the <**identity**> element in Web.config, or by calling the Win32® **LogonUser** API in your application code.

Important These approaches are not recommended. You should avoid them both on Windows 2000 servers, because it forces you to grant the "Act as part of the operating system" privilege to the ASP.NET process account. This significantly reduces the security of your Web application.
Windows Server 2003 will lift this restriction.

Using the Original Caller

To use the original caller's identity for remote resource access, you must be able to delegate the caller's security context from the Web server to the remote computer.

Scalability Warning: If you access the data services tier of your application using the original caller's impersonated identity, you severely impact the application's ability to scale, because database connection pooling is rendered ineffective. The security context for database connections is different for each user.

The following authentication schemes support delegation:

- **Kerberos**. For more information, see How To: Implement Kerberos Delegation for Windows 2000 within the Reference section of this guide.
- **Client certificates mapped to Windows accounts**. The mapping must be performed by IIS.
- **Basic**. Basic authentication supports remote resource access because the original caller's credentials are available in clear text at the Web server. These can be used to respond to authentication challenges from remote computers.

  Basic authentication must be used in conjunction with an interactive or batch logon session. The type of logon session that results from Basic authentication is configurable in the IIS Metabase. For more information, see the Platform SDK: Internet Information Services 5.1 on MSDN®.

  > **Important** Basic authentication is the least secure of the approaches that support delegation. This is because a clear text user name and password are passed from the browser to the server over the network and they are cached in memory at the Web server. You can use SSL to protect credentials while in transit but you should avoid caching clear text credentials at the Web server where possible.

To use the original caller for remote resource access

1. Configure IIS for Integrated Windows (Kerberos), Certificate (with IIS certificate mapping), or Basic authentication.

2. Configure ASP.NET for Windows authentication and impersonation.

```
3.      <authentication mode="Window" />

4.      <identity impersonate="true" />
```

5. If you use Kerberos delegation, configure Active Directory accounts for delegation.

More information

- For more information about configuring Kerberos delegation, see "How To: Implement Kerberos Delegation for Windows 2000" in the Reference section of this guide.

- For more information about IIS certificate mapping, see "Step-by-Step Guide to Mapping Certificates to User Accounts".
- For more information about ASP.NET Impersonation, see the .NET Framework Developers Guide on MSDN.

Accessing Files on a UNC File Share

If your application needs to access files on a Universal Naming Convention (UNC) share using ASP.NET, it is important to add NTFS permissions to the share's folder. You will also need to set the share's permissions to grant at least read access to either the ASP.NET process account or the impersonated identity (if your application is configured for impersonation).

Accessing Non-Windows Network Resources

If your application needs to access non-Windows resources such as databases located on non-Windows platforms or mainframe applications, you need to consider the following questions:

- What are the gatekeepers and trust boundaries associated with the resource?
- What credentials are required for authentication?
- Does the resource need to know the original caller identity, or does it trust the calling application (using a fixed process or service identity)?
- What is the performance cost associated with establishing connections? If the cost is significant you may need to implement connection pooling; for example, by using the object pooling feature of Enterprise Services.

If the resource needs to be able to authenticate the original caller (and Windows authentication is not an option), you have the following options:

- Pass credentials using (method call) parameters.
- Pass credentials in a connection string. Use SSL or IPSec to secure clear text credentials passed over a network.

  Store credentials securely within your application, for example by using DPAPI. For more information about securely storing database connection strings, see "Storing Database Connection Strings Securely" in Chapter 12, "Data Access Security."

- Use a centralized data store for authentication that both platforms can access; for example, an LDAP directory.

Secure Communication

Use SSL to secure the communication link between browser and Web server. SSL provides message confidentiality and message integrity. Use SSL and/or IPSec to provide a secure channel from Web server to application server or database server.

More information

For more information about secure communication, see "Chapter 4, Secure Communication.

Storing Secrets

Web applications often need to store secrets. These need to be secured against rogue administrators and malicious Web users, such as:

- **Rogue administrators**. Administrators and other unscrupulous users should not be able to view privileged information. For example, the administrator of the Web server should not be able to read the password of a SQL Server login account on a SQL Server computer located across the network.
- **Malicious Web users**. Even though there are components (such as the **FileAuthorizationModule**) that prevent users from accessing privileged files, if an attacker does gain access to a configuration file, the secret in the file should not be in plain text.

Typical examples of secrets include:

- **SQL connection strings**. A common mistake is to store the user name and password in plain text. The recommendation is to use Windows authentication instead of SQL authentication. If you can't use Windows authentication, see the following sections in Chapter 12, Data Access Security, which present secure alternatives:
  - Storing Database Connections Securely
  - Secure Communication
- **Credentials used for SQL application roles**. SQL Application roles must be activated with a stored procedure that requires the role name and associated password. For more information, see Authorization in Chapter 12, "Data Access Security."
- **Fixed identities in Web.config**. For example:

```
<identity impersonate="true" userName="bob"
  password="inClearText"/>
```

In the .NET Framework version 1.1, ASP.NET provides the ability to encrypt the username and password and store it safely in a registry key.

- **Process identity in Machine.config**. For example:

```
<process userName="cUsTuMUzerName" password="kUsTumPazzWerD" >
```

By default ASP.NET manages the secret if you use the "Machine" user name and "AutoGenerate" password.

In the .NET Framework version 1.1, ASP.NET provides the ability to encrypt the user name and password and store it safely in a registry key.

- **Keys used to store data securely**. It is impossible to safely store keys in software. However, certain tasks can mitigate the risk. An example is to create a custom configuration section handler, which uses asymmetric encryption to encrypt a session key. The session key can then be stored in a configuration file.
- **SQL Server session state**. To use SQL server to manage ASP.NET Web application session state, use the following Web.config settings.

```
<sessionState Ã‚Â… stateConnectionString="tcpip=127.0.0.1:42424"
                 sqlConnectionString="data source=127.0.0.1;
                 user id=UserName;password=MyPassword" />
```

In the .NET Framework 1.1, ASP.NET provides the ability to encrypt this information.

- **Passwords used for Forms authentication against a database**.

If your application validates authentication credentials against a database, don't store passwords in the database. Use a hash of the password with a salt value and compare hashes.

For more information, see "Authenticating Users against a Database" in Chapter 12, "Data Access Security."

Options for Storing Secrets in ASP.NET

A number of approaches are available to .NET Web application developers to store secrets. These include:

- **.NET cryptography classes**. The .NET Framework includes classes that can be used for encryption and decryption. These approaches require that you safely store the encryption key.
- **Data Protection API (DPAPI)**. DPAPI is a pair of Win32 APIs that encrypt and decrypt data by using a key derived from the user's password. When using DPAPI, you do not deal with key management. The operating system manages the key, which is the user's password.
- **COM+ Constructor Strings**. If your application uses serviced components, you can store the secret in an object construction string. The string is stored in the COM+ catalog in a clear text form,
- **CAPICOM**. This is a Microsoft COM object that provides COM-based access to the underlying Crypto API.
- **Crypto API**. These are low level Win32 APIs that perform encryption and decryption.

More information

For more information, see the entry for Cryptography, CryptoAPI and CAPICOM in the Platform SDK on MSDN.

Consider Storing Secrets in Files on Separate Logical Volumes

Consider installing Web application directories on a separate logical volume from the operating system (for example, E: instead of C:). This means that Machine.config (located under C:\WINNT\Microsoft.NET) and potentially other files that contain secrets such as, Universal Data Link (UDL) files, are located on a separate logical volume from the Web application directories.

The rationale for this approach is to protect against possible file canonicalization and directory traversal bugs because:

- File canonicalization bugs can expose files in the Web application folders.

  Note File canonicalization routines return the canonical form of a file path. This is usually the absolute pathname in which all relative references and references to the current directory have been completely resolved.

- Directory traversal bugs can expose files in other folders on the same logical volume.

No bugs of the sort described above have yet been published that exposed files on other logical volumes.

Securing Session and View State

Web applications must manage various types of state including view state and session state. This section discusses secure state management for ASP.NET Web applications.

Securing View State

If your ASP.NET Web applications use view state:

- Ensure the integrity of view state (to ensure it is not altered in any way while in transit) by setting the **enableViewStateMac** to true as shown below. This causes ASP.NET to generate a Message Authentication Code (MAC) on the page's view state when the page is posted back from the client.

```
<% @ Page enableViewStateMac=true >
```

- Configure the **validation** attribute on the <**machineKey**> element in Machine.config, to specify the type of encryption to use for data validation. Consider the following:
  - Secure Hash Algorithm 1 (SHA1) produces a larger hash size than Message Digest 5 (MD5) so it is considered more secure. However, view state protected with SHA1 or MD5 can be decoded in transit or on the client side and can potentially be viewed in plain text
  - Use 3 Data Encryption Standard (3DES) to detect changes in the view state and to also encrypt it while in transit. When in this state, even if view state is decoded, it cannot be viewed in plain text.

Securing Cookies

Cookies that contain authentication or authorization data or other sensitive data should be secured in transit by using SSL. For Forms authentication, the **FormsAuthentication.Encrypt** method can be used to encrypt the authentication ticket, passed between client and server in a cookie.

Securing SQL Session State

The default (in-process) ASP.NET session state handler has certain limitations. For example, it cannot work across computers in a Web farm. To overcome this limitation, ASP.NET allows session state to be stored in a SQL Server database.

SQL session state can be configured either in Machine.config or Web.config. The default setting in machine.config is shown below.

```
<sessionState mode="InProc"

              stateConnectionString="tcpip=127.0.0.1:42424"

              stateNetworkTimeout="10"

              sqlConnectionString="data source=127.0.0.1;user

                id=sa;password="

              cookieless="false" timeout="20"/>
```

By default, the SQL script InstallSqlState.sql, which is used for building the database used for SQL session state is installed at the following location:

```
C:\WINNT\Microsoft.NET\Framework\v1.0.3705
```

When you use SQL session state there are two problems to consider.

- You must secure the database connection string.
- You must secure the session state as it crosses the network.

Securing the Database Connection String

If you use SQL authentication to connect to the server, the user ID and password information is stored in plain text in web.config as shown below.

```
<sessionState

        cookieless="false"

        timeout="20"

        mode="InProc"

        stateConnectionString="tcpip=127.0.0.1:42424"

        sqlConnectionString=

          "data source=127.0.0.1;user

            id=UserName;password=ClearTxtPassword"
```

```
/>
```

By default the **HttpForbiddenHandler** protects configuration files from being downloaded. However, any user who has direct access to the folders where the configuration files are stored can still see the user name and password. A better practice is to use Windows authentication to SQL Server.

To use Windows authentication, you can use the ASP.NET process identity (typically ASPNET)

1. Create a duplicate account (with the same name and password) on the database server.
2. Create a SQL login for the account.
3. Create a database user in the **ASPState** database and map the SQL login to the new user.

   The **ASPState** database is created by the InstallSQLState.sql script.

4. Create a user defined database role and add the database user to the role.
5. Configure permissions in the database for the database role.

You can then change the connection string to use a trusted connection, as shown below:

```
sqlConnectionString="server=127.0.0.1;
                     database=StateDatabase;
                     Integrated Security=SSPI;"
```

Securing session state across the network

You may need to protect the session state as it crosses the network to the SQL Server database. This depends on how secure the network hosting the Web server and data servers is. If the database is physically secured in a trusted environment, you may be able to do without this extra security measure.

You can use IPSec to protect all IP traffic between the Web servers and SQL Server, or alternatively, you can use SSL to secure the link to SQL Server. With this approach, you have the option of encrypting just the connection used for the session state, and not all traffic that passes between the computers.

More information

- For more information about how to set up SQL Session State, see article Q317604, HOW TO: Configure SQL Server to Store ASP.NET Session State, in the Microsoft Knowledge Base.
- For more information about using SSL to SQL Server, see How To: Use SSL to Secure Communication with SQL Server 2000 in the Reference section of this guide.
- For more information about using IPSec, see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Web Farm Considerations

In a Web farm scenario, there is no guarantee that successive requests from the same client are serviced by the same Web server. This has implications for state management and for any encryption that relies on attributes maintained by the <**machineKey**> element in Machine.config.

Session State

The default ASP.NET in-process session state handling (which mirrors previous ASP functionality) results in server affinity and cannot be used in a Web farm scenario. For Web farm deployments, session state must be stored out of process in either the ASP.NET State service or a SQL Server database as described earlier.

Note You cannot rely on application state for maintaining global counters or unique values in Web farm (Web application configured to run on multiple servers) or Web garden (Web application configured to run on multiple processors) scenarios because application state is not shared across processes or computers.

DPAPI

DPAPI can work with either the machine store or user store (which requires a loaded user profile). If you use DPAPI with the machine store, the encrypted string is specific to a given computer and therefore you must generate the encrypted data on every computer. Do not copy the encrypted data across computers in a Web farm or cluster.

If you use DPAPI with the user store, you can decrypt the data on any computer with a roaming user profile.

More information

For more information about DPAPI, see Chapter 12, Data Access Security.

Using Forms Authentication in a Web Farm

If you are using Forms authentication, it is essential that all of the servers in the Web farm share a common machine key, which is used for encryption, decryption, and validation of the authentication ticket.

The machine key is maintained by the <**machineKey**> element within Machine.config. The default setting is shown below.

```
<machineKey validationKey="AutoGenerate"

            decryptionKey="AutoGenerate"

            validation="SHA1"/>
```

This setting results in every machine generating a different validation and decryption key. You must change the <**machineKey**> element and place common key values across all servers in the Web farm.

The <machineKey> Element

The <**machineKey**> element located in Machine.config is used to configure the keys used for encryption and decryption of Forms authentication cookie data and view state.

When the **FormsAuthentication.Encrypt** or **FormsAuthentication.Decrypt** methods are called, and when view state is created or retrieved, the values in the <**machineKey**> element are consulted.

```
<machineKey validationKey="autogenerate|value"

            decryptionKey="autogenerate|value"

            validation="SHA1|MD5|3DES" />
```

The validationKey attribute

The value of the **validationKey** attribute is used to create and validate MAC codes for view state and Forms authentication tickets. The validation attribute signifies what algorithm to use when performing the MAC generation. Note the following:

- With Forms authentication, this key works in conjunction with the <**forms**> **protection** attribute. When the protection attribute is set to **Validation**, and then when the **FormsAuthentication.Encrypt** method is called, the ticket value and the **validationKey** are used to compute a MAC that is appended to the cookie. When the **FormsAuthentication.Decrypt** method is called, the MAC is computed and compared to the MAC that is appended to the ticket.
- With view state, the value of a control's view state and the **validationKey** are used to compute a MAC, which is appended to the view state. When the view state is posted back from the client, the MAC is recomputed and compared to the MAC that is appended to the view state.

The decryptionKey attribute

The value of the **decryptionKey** attribute is used to encrypt and decrypt Forms authentication tickets and view state. The DES or Triple DES (3DES) algorithms are used. The precise algorithm depends on whether or not the Windows 2000 High Encryption Pack is installed on the server. If it is installed 3DES is used, otherwise DES is used. Note the following:

- With Forms authentication, the key works in conjunction with the <**forms**> **protection** attribute. When the **protection** attribute is set to **Encryption**, and the **FormsAuthentication.Encrypt** or **Decrypt** methods are called, the ticket value is encrypted or decrypted with the specified **decryptionKey** value.
- With view state, the value of a controls view state is encrypted with the **decryptionKey** value when sent to the client and is decrypted when the client posts the data back to the server.

The validation attribute

This attribute dictates what algorithm to use when validating, encrypting, or decrypting. It can take the values SHA1, MD5, or 3DES. The following describes these values:

- **SHA1**. The HMACSHA1 algorithm is actually used when the setting is SHA1. It produces a 160 bit (20 byte) hash or digest of the input. HMACSHA1 is a keyed hashing algorithm. The key used as the input for this algorithm is specified by the **validationKey** attribute.

  SHA1 is a popular algorithm because of its larger digest size compared to other algorithms.

- **MD5**. This produces a 20-byte hash using the MD5 algorithm.
- **3DES**. This encrypts data using the Triple DES (3DES) algorithm.

Note When the validation attribute is set to 3DES, it is not actually used by Forms authentication. SHA1 is used instead.

More information

- For information about how to create keys suitable for placing in Machine.config, see article Q312906, HOW TO: Create Keys w/ C# .NET for Use in Forms Authentication, in the Microsoft Knowledge Base.
- For more information about the Windows 2000 High Encryption Pack, see Windows 2000 High Encryption Pack.

Summary

This chapter has described a variety of techniques and approaches for securing ASP.NET Web applications. Much of the guidance and many of the recommendations presented in this chapter also apply to the development of ASP.NET Web services and .NET Remoting objects hosted by ASP.NET. To summarize:

- If your application uses Forms authentication and if performance is an issue when authenticating the user, retrieve a list of roles and store them in the authentication ticket.
- If you use Forms authentication, always create a principal and store it in the context on each request.
- If there are too many roles to store in an authentication cookie, then use the global application cache to store the roles.
- Don't create a custom least privileged account to run ASP.NET. Instead, change the ASPNET account password and create a duplicate account on any remote Windows server that your application needs to access.
- If you must create a custom account to run ASP.NET, use the principle of least privilege. For example:
  - Use a least privileged domain account if administration is the main concern.
  - If you use a local account. you must create a duplicated account on any remote computer that the Web application needs to access. You must use local accounts when your application needs to access resources in non-trusting domains, or where a firewall prevents Windows authentication.

- Don't run ASP.NET using the local SYSTEM account.
- Don't give the ASPNET account "Act as part of the operating system" privilege.

- Use SSL when:
  - Security sensitive information is passed between browser and Web server.
  - When Basic authentication is used (to protect credentials).
  - When Forms authentication is used for authentication (as opposed to personalization).
- Avoid storing secrets in plain text.

Enterprise Services Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter explains how to secure business functionality in serviced components contained within Enterprise Services applications. It shows you how and when to use Enterprise Services (COM+) roles for authorization, and how to configure RPC authentication and impersonation. It also shows you how to securely call serviced components from an ASP.NET Web application and how to identify and flow the original caller's security context through a middle tier serviced component. (30 printed pages)

Contents

Traditional COM+ services such as distributed transactions, just-in-time activation, object pooling, and concurrency management are available to .NET components. With .NET, such services are referred to as Enterprise Services. They are essential for many middle-tier .NET components running within .NET Web applications.

To add services to a .NET component, you must derive the component class from the **EnterpriseServices**.**ServicedComponent** base class and then specify precise service requirements using .NET attributes compiled into the assembly that hosts the component.

This chapter describes how to build secure serviced components and how to call them from ASP.NET Web applications.

Security Architecture

The authentication, authorization, and secure communication features supported by Enterprise Services applications are shown in Figure 9.1. The client application shown in Figure 9.1 is an ASP.NET Web application.

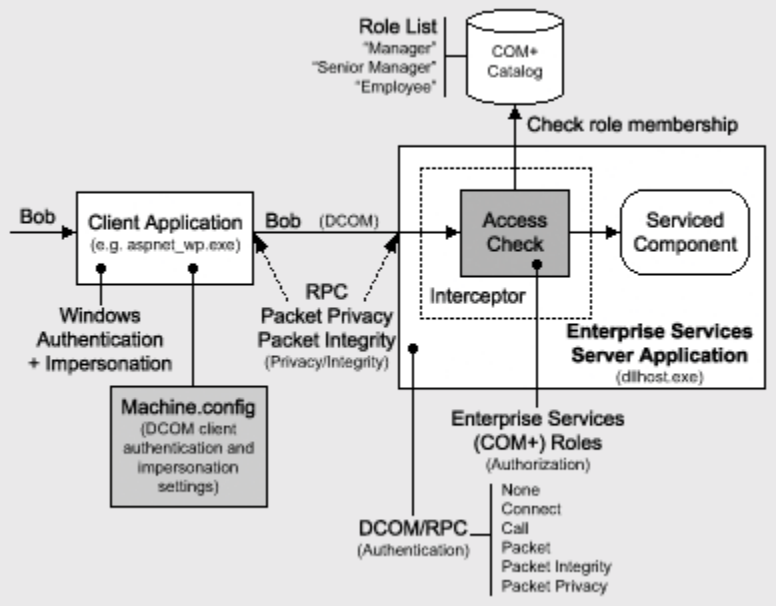

Figure 9.1. Enterprise Services role-based security architecture

Notice that authentication and secure communication features are provided by the underlying RPC transport used by Distributed COM (DCOM). Authorization is provided by Enterprise Services (COM+) roles.

The following summarizes the main elements of the Enterprise Services security architecture:

- Enterprise Services applications use RPC authentication to authenticate callers. This means that unless you have taken specific steps to disable authentication, the caller is authenticated using either Kerberos or NTLM authentication.
- Authorization is provided through Enterprise Services (COM+) roles, which can contain Microsoft® Windows® operating system group or user accounts. Role membership is defined within the COM+ catalog and is administered by using the Component Services tool.

  Note If the Enterprise Services application uses impersonation, caller authorization using Windows ACLs on secured resources is also available.

- When a client (for example, an ASP.NET Web application) calls a method on a serviced component, after the authentication process is complete, the Enterprise Services interception layer accesses the COM+ catalog to determine the client's role membership. It then checks whether membership of the role or roles permits authorized access to the current application, component, interface, and method.
- If the client's role membership permits access, the method is called. If the client doesn't belong to an appropriate role, the call is rejected, and a security event is optionally generated to reflect the failed access attempt.

  Important To implement meaningful role-based authorization within an Enterprise Services application called by an ASP.NET Web application, Windows authentication and impersonation must be used within the ASP.NET Web application in order to ensure that the original caller's security context flows through to the serviced component.

- To secure the DCOM communication link between client and server applications, either the RPC Packet Integrity authentication level can be used (to provide message integrity), or the RPC Packet Privacy authentication level can be used (to provide message confidentiality).

Gatekeepers and Gates

The Enterprise Services runtime acts as the gatekeeper for serviced components. The individual gates (authorization points) within an Enterprise Services application are shown in Figure 9.2. You configure these gates by using Enterprise Services roles, which you must populate with the appropriate Windows group and user accounts.

Note You must also ensure that access checking (role-based security) is enabled for your Enterprise Services application and that the appropriate level of authentication is being used. For more information about how to configure security, see Configuring Security later in this chapter.

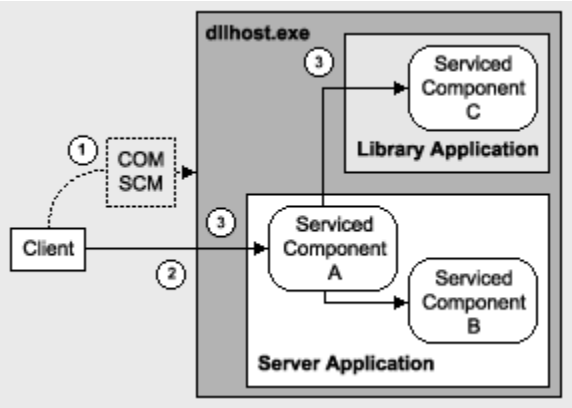

Figure 9.2. Gatekeepers within an Enterprise Services application

There are three distinct access checks performed in response to a client issuing a method call on a serviced component. These are illustrated in Figure 9.2 and described below:

1. An initial access check is performed by the subsystem responsible for activating Enterprise Services applications—the COM Service Control Manager (SCM)—when a call to a serviced component results in an activation request (and the creation of a new instance of the COM+ surrogate process, Dllhost.exe).

   To successfully pass this access check, the caller must be a member of at least one role defined within the application.

2. A second access check is performed when the client's call enters the Dllhost.exe process instance.

   Once again, if the caller is a member of at least one role defined within the application, this access check succeeds.

3. The final access check occurs when the client's call enters either a server or library application.

   To successfully pass this access check, the caller must be a member of a role that is associated with either, the interface, class, or method that is the target of the client's call.

   > **Important** After a call invokes a method on a serviced component, no further access checks are made if the component communicates with other components located in the same application. However, access checks do occur if a component calls another component within a separate application (library or server).

Use Server Applications for Increased Security

If your application needs to enforce an authentication level, for example because it requires encryption to ensure that the data sent to a serviced component remains confidential and tamper proof while in transit across the network, you should use a server application.

The authentication level can be enforced for a server application, while library applications inherit their authentication level from the host process.

To configure the activation type of an Enterprise Services application, use the assembly level **ApplicationActivation** attribute as shown below.

```
[assembly: ApplicationActivation(ActivationOption.Server)]
```

This is equivalent to setting the **Activation Type** to **Server application** on the **Activation** page of the application's **Properties** dialog within Component Services.

Security for Server and Library Applications

Role-based security works in a similar fashion for in-process library applications and out-of-process server applications.

Note the following differences for library applications:

- **Privileges**. The privileges of a library application are determined by the privileges of the client (host) process. For example, if the client process runs with administrator privileges, the library application will also have administrator privileges.
- **Impersonation**. The impersonation level of a library application is inherited from the client process and cannot be set explicitly.
- **Authentication**. The authentication level of a library application is inherited from the client process. With library applications, you can explicitly enable or disable authentication. This option is available on the **Security** page of a library application's **Properties** dialog box.

  This option is typically used to support unauthenticated call-backs from other out-of-process COM components.

Assign roles to classes, interfaces or methods

With library applications you should always assign roles at the class, interface or method level. This is also best practice for server applications.

Users that are defined within library application roles cannot be added to the security descriptor of the client process. This means that you must use at least class-level security to allow a library application to perform role-based authorization.

Code Access Security Requirements

Code Access Security (CAS) requires that code have particular permissions to be able to perform certain operations and access restricted resources. CAS is most useful in a client environment where code is downloaded from the Internet. In this type of situation it is unlikely that the code is fully trusted.

Typically, applications that use serviced components are fully trusted, and as a result CAS has limited use. However, Enterprise Services does demand that the calling code have the necessary permission to call unmanaged code. This implies the following:

- Unmanaged code permission is required to activate and perform cross context calls on serviced components.
- If the client of a serviced component is an ASP.NET Web application, this application must have unmanaged code permission.
- If a reference to a serviced component is passed to untrusted code, methods defined on the serviced component cannot be called from the untrusted code.

Configuring Security

This section shows you how to configure security for:

- A serviced component running in an Enterprise Services server (out-of-process) application.
- An ASP.NET Web application client.

Configuring a Server Application

The steps required to configure an Enterprise Services server application are shown in Figure 9.3.

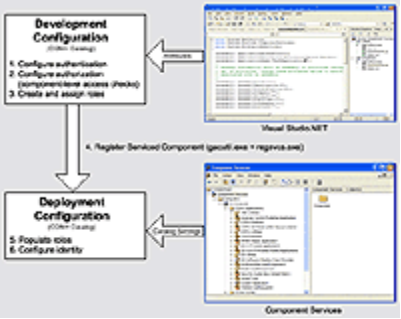

Figure 9.3. Configuring Enterprise Services security (click thumbnail for larger image)

Development time vs. deployment time configuration

You can configure most security settings within the COM+ catalog at development time by using .NET attributes within the assembly that contains the serviced component. These attributes are used to populate the COM+ catalog when the serviced component is registered with COM+ by using the Regsvcs.exe tool.

Other configuration steps such as populating roles with Windows group and user accounts and configuring a run-as identity for the server application (Dllhost.exe instance) must be configured using the Component Services administration tool (or programmatically using script) at deployment time.

Configure authentication

To set the application authentication level declaratively, use the **ApplicationAccessControl** assembly level attribute as shown below.

```
[assembly: ApplicationAccessControl(

              Authentication = AuthenticationOption.Call)]
```

This is equivalent to setting the **Authentication Level for Calls** value on the **Security** page of the application's **Properties** dialog within Component Services.

Note The client's authentication level also affects the authentication level used by the Enterprise Services application, because a process of high-water mark negotiation is employed, which always results in the higher of the two settings being used.
For more information about configuring the DCOM authentication level used by an ASP.NET client application, see Configuring an ASP.NET Client Application, later in this section.
For more information about DCOM authentication levels and authentication level negotiation, see the Security Concepts section of this chapter.

Configure authorization (component-level access checks)

To enable fine-grained authorization at the component, interface or method level you must:

- Enable access checks at the application level.

  Use the following .NET attribute to enable application-wide access checks.

  ```
  [assembly: ApplicationAccessControl(true)]
  ```

  This is equivalent to selecting the **Enforce access checks for this application** check box on the **Security** page of the application's **Properties** dialog box within Component Services.

  Important Failure to set this attribute results in no access checks being performed.

- Configure the application's security level at the process and component level.

  For meaningful role-based security, enable access checking at the process and component levels by using the following .NET attribute.

```
[assembly: ApplicationAccessControl(AccessChecksLevel=

                        AccessChecksLevelOption.

                          ApplicationComponent)]
```

  This is equivalent to selecting the **Perform access checks at the process and component levels** check box on the **Security** page of the application's **Properties** dialog box within Component Services.

  > **Note** Always enable access checking at the process and component level for library applications.

- Enable component-level access checks.

  To enable component-level access checks, use the **ComponentAccessControl** class-level attribute as shown below.

```
[ComponentAccessControl(true)]

public class MyServicedComponent : ServicedComponent

{

}
```

  This is equivalent to selecting the **Enforce Component Level Access Checks** check box on the **Security** page of the component **Properties** dialog box within Component Services.

  > **Note** This setting is effective only if you have enabled application-level access checking and have configured process and component level access checks, as described previously.

Create and assign roles

Roles can be created and assigned at the application, component (class), interface, and method levels.

Adding roles to an application

To add roles to an application, use the **SecurityRole** assembly level attribute as shown below.

```
[assembly:SecurityRole("Employee")]

[assembly:SecurityRole("Manager")]
```

This is equivalent to adding roles to an application by using the Component Services tool.

Note Using the **SecurityRole** attribute at the assembly level is equivalent to adding roles to the application, but not assigning them to individual components, interfaces, or methods. The result is that the members of these roles determine the composition of the security descriptor attached to the application. This is used solely to determine who is allowed to access (and launch) the application.
For more effective role-based authorization, always apply roles to components, interfaces, and methods as described below.

Adding roles to a component (class)

To add roles to a component apply the **SecurityRole** attribute above the class definition, as shown below.

```
 [SecurityRole("Manager")]

public class Transfer : ServicedComponent

{

}
```

Adding roles to an interface

To apply roles at the interface level, you must create an interface definition and then implement it within your serviced component class. You can then associate roles with the interface by using the **SecurityRole** attribute.

Important At development time, you must also annotate the class with the **SecureMethod** attribute. This informs Enterprise Services that method level security services may be used. At deployment time, administrators must also add users to the system defined **Marshaler** role, which is automatically created within the COM+ catalog, when a class that is marked with **SecureMethod** is registered with Component Services.
Use of the **Marshaler** role is discussed further in the next section.

The following example shows how to add the **Manager** role to a particular interface.

```
 [SecurityRole("Manager")]

public interface ISomeInterface

{

  void Method1( string message );

  void Method2( int parm1, int parm2 );

}



[ComponentAccessControl]

[SecureMethod]

public class MyServicedComponent : ServicedComponent, ISomeInterface

{

  public void Method1( string message )

  {

    // Implementation

  }

  public void Method2( int parm1, int parm2 )

  {

    // Implementation
```

```
  }
}
```

Adding roles to a method

To ensure that the public methods of a class appear in the COM+ catalog, you must explicitly implement an interface that defines the methods. Then, to secure the methods, you must use the **SecureMethod** attribute on the class, or the **SecureMethod** or **SecurityRole** attribute at the method level.

Note The **SecureMethod** and **SecurityRole** attributes must appear above the method implementation and not within the interface definition.

To enable method level security, perform the following steps:

1. Define an interface that contains the methods you want to secure. For example:

```
2.      public interface ISomeInterface
3.      {
4.        void Method1( string message );
5.        void Method2( int parm1, int parm2 );
6.      }
```

7. Implement the interface on the serviced component class:

```
8.       [ComponentAccessControl]
9.      public class MyServicedComponent : ServicedComponent,
10.       ISomeInterface
11.     {
12.       public void Method1( string message )
13.       {
14.         // Implementation
15.       }
16.       public void Method2( int parm1, int parm2 )
17.       {
18.         // Implementation
19.       }
20.     }
```

21. If you want to configure roles administratively by using the Component Services tool, you must annotate the class with the **SecureMethod** attribute, as shown below.

```
22.      [ComponentAccessControl]
23.      [SecureMethod]
24.     public class MyServicedComponent : ServicedComponent,
25.       ISomeInterface
26.     {
27.     }
```

28. Alternatively, if you want to add roles to methods at development time by using .NET attributes, apply the **SecurityRole** attribute at the method level. In this event, you do not need to apply the **SecureMethod** attribute at the class level (although the **ComponentAccessControl** attribute must still be present to configure component level access checks).

    In the following example only members of the **Manager** role can call **Method1**, while members of the **Manager** and **Employee** roles can call **Method2**.

```
 [ComponentAccessControl]
public class MyServicedComponent : ServicedComponent,
  ISomeInterface
{
  [SecurityRole("Manager")]
  public void Method1( string message )
  {
    // Implementation
  }
  [SecurityRole("Manager")]
  [SecurityRole("Employee")]
  public void Method2( int parm1, int parm2 )
  {
    // Implementation
  }
}
```

29. At deployment time, administrators must add any user that requires access to methods or interfaces of the class to the predefined **Marshaler** role.

    Note The Enterprise Services infrastructure uses a number of system-level interfaces that are exposed by all serviced components. These include **IManagedObject**, **IDisposable**, and **IServiceComponentInfo**. If

access checks are enabled at the interface or method levels, the Enterprise Services infrastructure is denied access to these interfaces.

As a result, Enterprise Services creates a special role called **Marshaler** and associates the role with these interfaces. You can view this role (and the aforementioned interfaces) with the Component Services tool.

At deployment time, application administrators need to add all users to the **Marshaler** role who needs to access any methods or interface of the class. You can automate this in two different ways:

- Write a script that uses the Component Services object model to copy all users from other roles to the Marshaler role.
- Write a script that assigns all other roles to these three special interfaces and delete the Marshaler role.

Register serviced components

Register serviced components in:

- **The Global Assembly Cache**. Serviced components hosted in COM+ server applications require installation in the global assembly cache, while library applications do not.

  To register a serviced component in the global assembly cache, run the Gacutil.exe command line utility. To register an assembly called MyServicedComponent.dll in the global assembly cache, run the following command.

  ```
  Gacutil-i MyServicedComponent.dll
  ```

  Note You can also use the Microsoft .NET Framework Configuration Tool from the **Administrative Tools** program group to view and manipulate the contents of the global assembly cache.

- **The COM+ Catalog**. To register an assembly called MyServicedComponent.dll in the COM+ catalog, run the following command.

```
regsvcs.exe MyServicedComponent.dll
```

This command results in the creation of a COM+ application. The .NET attributes present within the assembly are used to populate the COM+ catalog.

Populate roles

Populate roles by using the Component Services tool, or by using script to program the COM+ catalog using the COM+ administration objects.

Use Windows groups

Add Windows 2000 group accounts to Enterprise Services roles for maximum flexibility. By using Windows groups, you can effectively use one administration tool (the Users and Computers Administration tool) to administer both Windows and Enterprise Services security.

- Create a Windows group for each role in the Enterprise Services application.
- Assign each group to its respective role.

For example, if you have a role called **Manager**, create a Windows group called **Managers**. Assign the **Managers** group to the **Manager** role.

- After you assign groups to roles, use the Users and Computers Administration tool to add and remove users in each group.

  For example, adding a Windows 2000 user account named **David** to the Windows 2000 group Managers effectively maps **David** to the **Manager** role.

To assign Windows groups to Enterprise Services roles by using Component Services

1. Using the Component Services tool, expand the application that contains the roles to which you want to add Windows 2000 groups.
2. Expand the **Roles** folder and the specific role to which you want to assign Windows groups.
3. Select the **Users** folder under the specific role.
4. Right-click the folder, point to **New,** and then click **User**.
5. In the **Select Users or Groups** dialog box, add groups (or users) to the role.

More information

For more information about programming the COM+ catalog by using the COM+ administration objects, see Automating COM+ Administration within the Component Development section of the MSDN Library.

Configure identity

Use the Component Services tool (or script) to configure the identity of the Enterprise Services application. The identity property determines the account used to run the instance of Dllhost.exe that hosts the application.

To configure identity

1. Using the Component Services tool, select the relevant application.
2. Right-click the name of the application, and then click **Properties**.
3. Click the **Identity** tab.
4. Click **This user** and specify the configured service account used to run the application.

More information

For more information about choosing an appropriate identity to run an Enterprise Services application, see Choosing a Process Identity later in this chapter.

Configuring an ASP.NET Client Application

You must configure the DCOM authentication level and impersonation levels used by client applications when communicating with serviced components using DCOM.

Configure authentication

To configure the default authentication level used by an ASP.NET Web application when it communicates with a serviced component, edit the **comAuthenticationLevel** attribute on the <**processModel**> element in Machine.config.

Machine.config is located in the following folder.

```
%windir%\Microsoft.NET\Framework\v1.0.3705\CONFIG
```

Set the **comAuthenticationLevel** attribute to one of the following values.

```
comAuthenticationLevel=

            "[Default|None|Connect|Call|Pkt|PktIntegrity|PktPrivacy]"
```

More information

For more information about DCOM authentication levels, see Authentication within the "Security Concepts" section later in this chapter.

Configure impersonation

The impersonation level set by the client determines the impersonation level capabilities of the server. To configure the default impersonation level used by a Web-based application when it communicates with a serviced component, edit the **comImpersonationLevel** attribute on the <**processModel**> element in Machine.config. Set it to one of the following values.

```
comImpersonationLevel="[Default|Anonymous|Identify|Impersonate|Delega

  te]"
```

More information

For more information about DCOM impersonation levels, see Impersonation within the "Security Concepts" section later in this chapter.

Configuring Impersonation Levels for an Enterprise Services Application

If a serviced component in one application needs to call a serviced component within a second (server) application, you may need to configure the impersonation level for the client application.

Important The impersonation level configured for an Enterprise Services application (on the **Security** page of the application's **Properties** dialog box) is the impersonation level used by outgoing calls made by components within the application. It does not affect whether or not serviced components within the application perform impersonation. To impersonate clients within a serviced component, you must use programmatic impersonation techniques, as described in "Flowing the Original Caller," later in this chapter.

To set the application impersonation level declaratively, use the **ApplicationAccessControl** assembly level attribute as shown below.

```
 [assembly: ApplicationAccessControl(

              ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

This is equivalent to setting the **Impersonation Level** value on the **Security** page of the application's **Properties** dialog within Component Services.

Programming Security

The Enterprise Services security features are available to .NET components using the **ContextUtil**, **SecurityCallContext**, and **SecurityIdentity** classes.

Programmatic Role-Based Security

For fine-grained authorization decisions, you can programmatically test role membership using the **IsCallerInRole** method of the **ContextUtil** class. Prior to calling this method, always check that component-level access checks are enabled, as shown in the following code fragment. If security is disabled, **IsCallerInRole** always returns true.

```
public void Transfer(string fromAccount, string toAccount, double

  amount)

{

  // Check that security is enabled

  if (ContextUtil.IsSecurityEnabled)
```

```
  {
    // Only Managers are allowed to transfer sums of money in excess
    // of $1000
    if (amount > 1000)
    {
      if (ContextUtil.IsCallerInRole("Manager"))
      {
        // Caller is authorized
      }
      else
      {
        // Caller is unauthorized
      }
    }
}
```

Identifying Callers

The following example shows how to identify all upstream callers from within a serviced component.

```
 [ComponentAccessControl]
public class MyServicedComponent : ServicedComponent
{
  public void ShowCallers()
  {
    SecurityCallContext context = SecurityCallContext.CurrentCall;
    SecurityCallers callers = context.Callers;
    foreach(SecurityIdentity id in callers)
    {
      Console.WriteLine(id.AccountName);
    }
```

```
  }

}
```

Note The original caller identity is available via the **SecurityCallContext.OriginalCaller** property.

Choosing a Process Identity

Server activated Enterprise Services applications run within an instance of the Dllhost.exe process. You must configure the account used to run the process in the COM+ catalog by using the Component Services tool.

Note You cannot specify the run as identity by using a .NET attribute.

Never Run as the Interactive User

Do not run server applications using the identity of the interactively logged on user (this is the default setting). There are two main reasons to avoid this:

- The privileges and access rights of the application will vary and will be dependent upon who is currently logged on interactively at the server. If an administrator happens to be logged on, the application will have administrator privileges.
- If the application is launched while a user is interactively logged on and then the user logs off, the server application will be shut down. It will not be able to restart until another user logs on interactively.

The interactive user setting is designed for developers to use at development time, and should not be considered a deployment setting.

Use a Least-Privileged Custom Account

Create a least privileged account to mitigate the threat associated with a process compromise. If a determined attacker manages to compromise the server process, he or she will easily be able to inherit the privileges and access rights granted to the process account. An account configured with minimum privileges restricts the potential damage that can be done.

If you need to access network resources with the process account, the remote computer must be able to authenticate the process account. In this scenario, you have two options:

- You can use a domain account if the two computers are in the same or trusting domains.
- You can use a local account and then create a duplicate account (with the same user name and password) on the remote computer. With this option, you must ensure that the passwords of the two accounts remain synchronized.

  You may be forced to use the duplicated local account approach if the remote computer is located in a separate domain (with no trust relationship), or if the remote computer is behind a firewall (where closed ports do not permit Windows authentication).

Accessing Network Resources

Your serviced components may need to access remote resources. It is important to be able to identify the following:

- The resources the components need to access. For example, files on file shares, databases, other DCOM servers, Active Directory® directory service objects, and so on.
- The identity used to perform the resource access. If your serviced component accesses remote resources, the identity used (which by default is the process identity) must be capable of being authenticated by the remote computer.

  Note For information specific to accessing remote SQL Server databases, see Chapter 12, Data Access Security.

You can access remote resources from a component within an Enterprise Services application by using any of the following identities:

- The original caller (if you are explicitly impersonating by using **CoImpersonateClient**)
- The current process identity (configured in the COM+ catalog for server applications)
- A specific service account

Using the Original Caller

To use the original caller's identity for remote resource access, you must:

- Programmatically impersonate the original caller by calling **CoImpersonateClient**.
- Be able to delegate the caller's security context from the application server hosting the Enterprise Services application to the remote computer. This assumes that you are using Kerberos authentication between your Enterprise Services application and client application.

Scalability Warning If you access the data services tier of your application using the original caller's impersonated identity, you severely impact the application's ability to scale, because you prevent database connection pooling from working efficiently; it doesn't work efficiently because the security context of each database connection is tied to many individual callers.

More information

For more information about impersonating callers, see Flowing the Original Caller, later in this chapter.

Using the Current Process Identity

If your application is configured to run as a server application, you can use the configured process identity for remote resource access (this is the default case).

If you want to use the server process account for remote resource access, you must either:

- Run the server application using a least-privileged domain account. This assumes that client and server computers are in the same or trusting domains.
- Duplicate the process account using the same username and password on the remote computer.

If ease of administration is your primary concern, you should use a least-privileged domain account.

If your application is configured to run as a library application, the process identity is inherited from the host process (which will often be a Web-based application). For more information about using the ASP.NET process identity for remote resource access, see Chapter 8, ASP.NET Security.

Using a Specific Service Account

Your Enterprise Services application could access remote resources by using a specifically configured service account (that is, a non-user Windows account). However, this approach is not recommended on Windows 2000 because it relies on you calling the **LogonUser** API.

The use of **LogonUser** on Windows 2000, forces you to grant the "Act as part of the operating system" privilege to the Enterprise Services process account. This significantly reduces the security of your application.

Note Microsoft Windows Server 2003 will lift this restriction.

Flowing the Original Caller

By default, outgoing calls issued by serviced components (for example, to access local or remote resources) are made using the security context obtained from the host process. For server applications, this is the configured run-as identity. For library applications, this is the identity of the (host) client process (for example, Aspnet_wp.exe when an ASP.NET Web application is the client).

To flow the original caller's context through an Enterprise Services application

1. Call **CoImpersonateClient**.

   This creates and attaches a thread impersonation token to the current thread.

2. Perform operation (access local or remote resource).

   As impersonation is enabled, the outgoing call is made using the client's security context (as defined by the impersonation token).

   If local resources are accessed, the caller (client process) must have specified at least Impersonate level impersonation. If remote resources are accessed, the caller must have specified Delegate level impersonation.

   If the caller is an ASP.NET Web application, the default impersonation level for the ASP.NET worker process is Impersonate. Therefore, to flow the original caller to a downstream remote computer, you must change this default to Delegate (on the <**processModel**> element of Machine.config on the client computer).

   > **Note** To use the original caller's security context to access remote resources you must use Kerberos authentication, with accounts configured for delegation. The account used to run the Enterprise Services server application must also be marked in Active Directory as "Trusted for delegation."

3. Cease impersonation by calling **CoRevertToSelf**.

   This removes the impersonation token. Any subsequent call from the current method uses the process security context. If you fail to call **CoRevertToSelf**, it is called implicitly by the runtime when the method ends.

Note The identity of the original caller automatically flows to an Enterprise Services application and is available using **SecurityCallContext.OriginalCaller**. This can be useful for auditing purposes.

Calling CoImpersonateClient

CoImpersonateClient (and **CoRevertToSelf**) are located within OLE32.dll. You must import their definitions by using the **DllImport** attribute in order to be able to call them through P/Invoke. This is illustrated in the following code fragment.

```
class COMSec

{

  [DllImport("OLE32.DLL", CharSet=CharSet.Auto)]

  public static extern uint CoImpersonateClient();



  [DllImport("OLE32.DLL", CharSet=CharSet.Auto)]

  public static extern uint CoRevertToSelf();

}

. . .

void SomeMethod()

{

  // To flow the original caller's security context and use it to
```

```
  // access local or remote resources, start impersonation

  COMSec.CoImpersonateClient();

  // Perform operations as the caller

  // Code here uses the context of the caller - not the context of

  // the process

  . . .

  COMSec.CoRevertToSelf();

  // Code here reverts to using the process context

}
```

More information

For more information about how to configure a complete Kerberos delegation scenario that shows how to flow the original caller's security context through an ASP.NET Web application, an Enterprise Services application, and onto a database, see Flowing the Original Caller to the Database in Chapter 5, Intranet Security.

RPC Encryption

To secure the data sent from a client application to a remote serviced component over DCOM, use the RPC Packet Privacy authentication level between client and server. This provides message confidentiality and integrity.

You must configure the authentication level at the client and server.

To configure ASP.NET (where an ASP.NET Web application is the client), set the **comAuthenticationLevel** attribute on the <**processModel**> element in machine.config to **PktPrivacy**.

To configure an Enterprise Services server application, set the application-level authentication level either by using the Component Services tool or the following .NET attribute within the serviced component assembly.

```
[assembly: ApplicationAccessControl(

              Authentication = AuthenticationOption.Privacy)]
```

More Information

- For more information about configuring security (including authentication levels), see Configuring Security earlier in this chapter.
- For more information about RPC/DCOM authentication levels, see Authentication later in this chapter.
- For more information about authentication-level negotiation, see Authentication Level Negotiation later in this chapter.

Building Serviced Components

For a step-by-step walkthrough that shows you how to build a serviced component, see How To: Use Role-based Security with Enterprise Services in the Reference section of this guide.

DLL Locking Problems

When you rebuild a serviced component, if the DLL is locked:

- Use Component Services to shut down the COM+ server application.
- If you are developing a library application, the application may still be loaded into the Aspnet_wp.exe process. Run **IISReset** from a command prompt or use Task Manager to stop the Aspnet_wp.exe process.
- Use the FileMon.exe tool from www.sysinternals.com to help troubleshoot file locking problems.

Versioning

The default **AssemblyVersion** attribute that is generated by Microsoft Visual Studio® .NET development system when you create a new project is shown below.

```
[assembly: AssemblyVersion("1.0.*")]
```

Each time you rebuild the project, a new assembly version is generated. This also results in the generation of a new class identifier (CLSID) to identify the serviced component classes. If you repeatedly register the assembly with component services using Regsvcs.exe, you will see duplicated components (strictly classes) with different CLSIDs listed beneath the **Components** folder.

While this complies with strict COM versioning semantics and will prevent existing managed and unmanaged clients from breaking, it can be an annoyance during development.

During test and development, consider setting an explicit version by using the assembly level **AssemblyVersion** attribute shown below.

```
 [assembly: AssemblyVersion("1.0.0.1")]
```

This setting will prevent a new CLSID being generated with each successive project build. You may also want to fix the interface identifiers (IIDs). If your class implements explicit interfaces, you can fix the IID for a given interface by using the GUID attribute as shown below.

```
 [Guid("E1FBF27E-9F11-474d-8DF6-58916F798E9D")]

public interface IMyInterface

{

}
```

To generate new GUIDs

1. On the **Tools** menu of Visual Studio .NET, click **Create GUID**.
2. Click **Registry Format.**
3. Click **New GUID**.
4. Click **Copy**.
5. Paste the GUID from the clipboard into your source code.

> **Important** Prior to deploying your serviced component assembly for test and production, remove any fixed GUIDs and revert to an automated assembly versioning mechanism (for example, by using "1.0.*"). Failure to do so increases the likelihood that a new release of your component will break existing clients.

More information

For more information about versioning for deployment, see Understanding Enterprise Services (COM+) in .NET on MSDN.

QueryInterface Exceptions

If you see a **QueryInterface** call for the **IRoleSecurity** interface failing, this indicates that you have updated an interface definition within your assembly, but have not re-registered the assembly with Component Services using Regsvcs.exe.

Important Each time you run Regsvcs.exe you will need to reconfigure a server application's run-as identity and will also need to add users to groups again. You can create a simple script to automate this task.

DCOM and Firewalls

Windows 2000 (SP3 or QFE 18.1) or Windows Server 2003 allow you to configure Enterprise Services applications to use a static endpoint. If a firewall separates the client from the server, you only need to open two ports in the firewall. Specifically, you must open port 135 for RPC and a port for your Enterprise Services application.

As an alternative to this approach consider exposing your Enterprise Services application as a Web service. This allows you to activate and call serviced components by using SOAP over port 80. The main issue with this approach is that it doesn't allow you to flow transaction context from client to server. You would need to initiate your transaction at the remote serviced component.

More Information

For more information, see the following Knowledge Base articles:

- Article Q312960, Cannot Set Fixed Endpoint for a COM+ Application
- Article Q259011, SAMPLE: A Simple DCOM Client Server Test Application
- Article Q248809, PRB: DCOM Does Not Work over NAT-Based Firewall
- Article Q250367, INFO: Configuring Microsoft Distributed Transaction Coordinator (DTC) to Work Through a Firewall
- Article Q154596, HOWTO: Configure RPC Dynamic Port Allocation to Work w/ Firewall

Calling Serviced Components from ASP.NET

This section highlights the main issues you will encounter when an ASP.NET application calls a serviced component.

Caller's Identity

When you call a serviced component from an ASP.NET application, the security identity for the call is obtained from the application's Win32® thread identity. If the Web application is configured to impersonate the caller, this is the caller's identity. Otherwise, this is the ASP.NET process identity (by default, ASPNET).

From an ASP.NET application, you can retrieve the current Win32 thread identity by calling **WindowsIdentity.GetCurrent()**.

From a serviced component, you can retrieve the original caller identity by using **SecurityCallContext.OriginalCaller**.

Use Windows Authentication and Impersonation Within the Web-based Application

To enable meaningful role-based security within your Enterprise Services application, you must use Windows authentication and enable impersonation. This ensures that the serviced components are able to authenticate the original callers and make authorization decisions based on the original caller's identity.

Configure Authentication and Impersonation within Machine.config

DCOM authentication levels are negotiated between client (for example, the Web-based application) and server (the Enterprise Services application). The higher of the two security settings is used.

Configure ASP.NET authentication levels by using the **comAuthenitcation** attribute on the <**processModel**> element of Machine.config.

Impersonation levels are controlled by the client (for example, a Web-based application). The client can determine the degree of impersonation that it is willing to allow the server to use.

Configure ASP.NET impersonation levels (for all outgoing DCOM calls), by using the **comImpersonationLevel** attribute on the <**processModel**> element of Machine.config.

Configuring Interface Proxies

The security settings that apply to individual interface proxies are usually obtained from the default process level security settings. In the case of ASP.NET, default security settings such as the impersonation level and authentication level are configured in Machine.config, as described earlier.

If necessary, you can alter the security settings used by an individual interface proxy. For example, if your ASP.NET application communicates with a serviced component that exposes two interfaces and sensitive data is passed through only one interface, you may choose to use the encryption support provided by the packet privacy authentication level only on the sensitive interface and to use, for example, packet authentication on the other interface. This means that you do not experience the performance hit associated with encryption on both interfaces.

Collectively, the set of security settings that apply to an interface proxy are referred to as the security blanket. COM provides the following functions to allow you to query and manipulate security blanket settings on an individual interface proxy:

- CoQueryProxyBlanket
- CoSetProxyBlanket
- CoCopyProxy

You must use P/Invoke to call these functions from an ASP.NET Web application (the DCOM client), The following code shows how to configure a specific interface to use the Packet Privacy authentication level (which provides encryption). This code can be used from an ASP.NET Web application that communicates with a remote serviced component.

```
// Define a wrapper class for the P/Invoke call to CoSetProxyBlanket

class COMSec

{

  // Constants required for the call to CoSetProxyBlanket

  public const uint RPC_C_AUTHN_DEFAULT           = 0xFFFFFFFF;

  public const uint RPC_C_AUTHZ_DEFAULT           = 0xFFFFFFFF;

  public const uint RPC_C_AUTHN_LEVEL_PKT_PRIVACY = 6;

  public const uint RPC_C_IMP_LEVEL_DEFAULT       = 0;

  public const uint COLE_DEFAULT_AUTHINFO         = 0xFFFFFFFF;

  public const uint COLE_DEFAULT_PRINCIPAL        = 0;

  public const uint EOAC_DEFAULT                  = 0x800;


  // HRESULT  CoSetProxyBlanket( IUnknown * pProxy,
```

```
  //                              DWORD dwAuthnSvc,
  //                              DWORD dwAuthzSvc,
  //                              WCHAR * pServerPrincName,
  //                              DWORD dwAuthnLevel,
  //                              DWORD dwImpLevel,
  //                              RPC_AUTH_IDENTITY_HANDLE pAuthInfo,
  //                              DWORD dwCapabilities );
[DllImport("OLE32.DLL", CharSet=CharSet.Auto)]
public unsafe static extern uint CoSetProxyBlanket(
                                      IntPtr pProxy,
                                      uint dwAuthnSvc,
                                      uint dwAuthzSvc,
                                      IntPtr pServerPrincName,
                                      uint dwAuthnLevel,
                                      uint dwImpLevel,
                                      IntPtr pAuthInfo,
                                      uint dwCapababilities);
} // end class COMSec



// Code to call CoSetProxyBlanket
void CallComponent()
{
  // This is the interface to configure
  Guid IID_ISecureInterface = new Guid("c720ff19-bec1-352c-bb4b-
    e2de10b858ba");
  IntPtr pISecureInterface;
```

```
// Instantiate the serviced component
CreditCardComponent comp = new CreditCardComponent();
// Get its IUnknown pointer
IntPtr pIUnk = Marshal.GetIUnknownForObject(comp);
// Get the interface to configure
Marshal.QueryInterface(pIUnk, ref IID_ISecureInterface,
                       out pISecureInterface);
try
{
  // Configure the interface proxy and set packet privacy
  //authentication
  uint hr = COMSec.CoSetProxyBlanket( pISecureInterface,
                                      COMSec.RPC_C_AUTHN_DEFAULT,
                                      COMSec.RPC_C_AUTHZ_DEFAULT,
                                      IntPtr.Zero,
                                COMSec.RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
                                COMSec.RPC_C_IMP_LEVEL_DEFAULT,
                                      IntPtr.Zero,
                                      COMSec.EOAC_DEFAULT );
  ISecureInterface secure = (ISecureInterface)comp;
  // The following call will be encrypted as ISecureInterface is
  // configured for packet privacy authentication. Other interfaces
  // use the process level defaults (normally packet
  // authentication).
  secure.ValidateCreditCard("123456789");
}
catch (Exception ex)
{
```

```
  }

}
```

More information

- For more information about configuring an ASP.NET client application to call serviced components, see Configuring an ASP.NET Client Application, earlier in this chapter.
- For more information about DCOM authentication levels, see Authentication, later in this chapter.
- For more information about DCOM impersonation levels, see Impersonation, later in this chapter.
- For more information about using Windows authentication and enabling impersonation within a Web-based application, see Chapter 8, ASP.NET Security.

Security Concepts

This section provides a brief overview of Enterprise Services security concepts. If you are already experienced with COM+, many of the concepts will be familiar.

For background information on Enterprise Services, see the MSDN article Understanding Enterprise Services (COM+) in .NET.

The following are summaries of key security concepts that you should understand:

- Security settings for serviced components and Enterprise Services applications are maintained within the COM+ catalog. Most settings can be configured using .NET attributes. All settings can be configured by using the Component Services administration tool or Microsoft Visual Basic® Scripting Edition development system scripts.
- Authorization is provided by Enterprise Services (COM+) roles, which can contain Windows group or user accounts. These are not the same as .NET roles.
  - Role-based security can be applied at the application, interface, class, and method levels.
  - Imperative role checks can be performed programmatically within methods by using the **IsCallerInRole** method of the **ContextUtil** class.
- Effective role-based authorization within an Enterprise Services application relies on a Windows identity being used to call serviced components.
  - This may require you to use Windows authentication coupled with impersonation within an ASP.NET Web application—if the Web application calls serviced components that rely on Enterprise Services (COM+) roles.
  - When you call a serviced component from an ASP.NET Web application or Web service, the identity used for the outgoing DCOM call is determined by the Win32 thread identity as defined by **WindowsIdentity.GetCurrent()**.
- Serviced components can run in server or library applications.
  - Server applications run in separate instances of Dllhost.exe.
  - Library applications run in the client's process address space.
  - Role-based authorization works in a similar fashion for server and library applications, although there are some subtle differences between library and server applications from a security perspective. For details, see "Security for Server and Library Applications" earlier in this chapter.
- Authentication is provided by the underlying services of DCOM and RPC. The client and server's authentication level combined to determine the resulting authentication level used for communication with the serviced component.
- Impersonation is configured within the client application. It determines the impersonation capabilities of the server.

Enterprise Services (COM+) Roles and .NET Roles

Enterprise Services (COM+) roles are used to represent common categories of users who share the same security privileges within an application. While conceptually similar to .NET roles, they are completely independent.

Enterprise Services (COM+) roles contain Windows user and group accounts (unlike .NET roles that can contain arbitrary non-Windows user identities). Because of this, Enterprise Services (COM+) roles are only an effective authorization mechanism for applications that use Windows authentication and impersonation (in order to flow the caller's security context to the Enterprise Services application).

Table 9.1. Comparing Enterprise Services (COM+) roles with .NET roles

Feature	Enterprise Services (COM+) Roles	.NET Roles
Administration	Component Services Administration Tool	Custom
Data Store	COM+ Catalog	Custom data store (for example, SQL Server or Active Directory)
Declarative	Yes [SecurityRole("Manager")]	Yes [PrincipalPermission(SecurityAction.Demand, Role="Manager")]
Imperative	Yes ContextUtil.IsCallerInRole()	Yes IPrincipal.IsInRole
Class, Interface, and Method Level Granularity	Yes	Yes
Extensible	No	Yes (using custom IPrincipal implementation)
Available to all .NET components	Only for components that derive from ServicedComponent base class	Yes
Role Membership	Roles contain Windows group or user accounts	When using WindowsPrincipals, roles ARE Windows groups—no extra level of abstraction
Requires explicit Interface implementation	Yes To obtain method level authorization, an interface must be explicitly defined and implemented	No

Authentication

Because Enterprise Services rely on the underlying infrastructure provided by COM+ and DCOM/RPC, the authentication level settings available to Enterprise Services applications are those defined by RPC (and used by DCOM).

Table 9.2. Enterprise Services applications authentication settings

Authentication Level	Description
Default	Choose authentication level using normal negotiation rules
None	No authentication
Connect	Only authenticate credentials when the client initially connects to the server
Call	Authenticate at the start of each remote procedure call

Packet	Authenticate all data received from the client
Packet Integrity	Authenticate all data and verify that none of the transferred data has been modified
Packet Privacy	Authenticate all data and encrypt parameter state for each remote procedure call

Authentication level promotion

You should be aware that certain authentication levels are silently promoted. For example:

- If the User Data Protocol (UDP) datagram transport is used, Connect and Call levels are promoted to Packet, because the aforementioned authentication levels only make sense over a connection oriented transport such as TCP.

  Note Windows 2000 defaults to RPC over TCP for DCOM communications.

- For inter-process calls on a single computer, all authentication levels are always promoted to Packet Privacy. However, in a single computer scenario, data is not encrypted for confidentiality (because the data doesn't cross the network).

Authentication level negotiation

The authentication level used by Enterprise Services to authenticate a client is determined by two settings:

- **The process level authentication level**. For a server-activated application (running within Dllhost.exe), the authentication level is configured within the COM+ catalog.
- **The client authentication level**. The configured authentication level of the client process that communicates with the serviced component also affects the authentication level that is used.

  The default authentication level for an ASP.NET Web application is defined by the **comAuthenticationLevel** attribute on the <**processModel**> element in Machine.config.

The higher of the two (client and server) authentication level is always chosen. This is illustrated in the Figure 9.4.

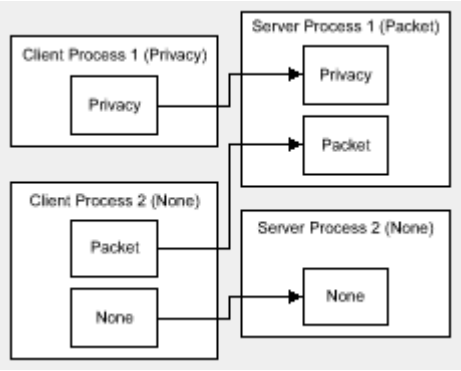

Figure 9.4. Authentication level negotiation

More information

For information about how to configure authentication levels for an Enterprise Service application, see Configuring Security earlier in this chapter.

Impersonation

The impersonation level defined for an Enterprise Services application determines the impersonation level to be used for all outgoing DCOM calls made by serviced components within the application.

Important It does NOT determine whether or the not serviced components within the application impersonate their callers. By default, serviced components do not impersonate callers. To do so, the service component must call **CoImpersonateClient**, as described in "Flowing the Original Caller" earlier in this chapter.

Impersonation is a client-side setting. It offers a degree of protection to the client as it allows the client to restrict the impersonation capabilities of the server.

Table 9.3. Available impersonation levels

Impersonation Level	Description
Identify	Allows the server to identify the client and perform access checks using the client's access token
Impersonate	Allows the server to access local resources using the client's credentials
Delegate	Allows the server to access remote resources using the client's credentials (this requires Kerberos and specific account configuration)

The default impersonation level used by a Web-based application when it communicates with serviced components (or any component using DCOM) is determined by the **comImpersonationLevel** attribute on the <**processModel**> element in Machine.config.

Cloaking

Cloaking determines precisely how client identity is projected through a COM object proxy to a server during impersonation. There are two forms of cloaking:

- **Dynamic Cloaking**. Enterprise Services server applications use dynamic cloaking (this is not configurable). Cloaking for library applications is determined by the host process, for example the ASP.NET worker process (Aspnet_wp.exe). Web-based applications also use dynamic cloaking—again this is not configurable.

  Dynamic cloaking causes the thread impersonation token to be used to represent the client's identity during impersonation. This means that if you call **CoImpersonateClient** within a serviced component, the client's identity is assumed for subsequent outgoing calls made by the same method, until either **CoRevertToSelf** is called or the method ends (where **CoRevertToSelf** is implicitly called).

- **Static Cloaking**. With static cloaking, the server sees the credentials that are used on the first call from client to server (irrespective of whether or not a thread is impersonating during an outgoing call).

More information

- For information about how to configure impersonation levels for Enterprise Service applications, see Configuring Security, earlier in this chapter.
- For more information about cloaking, see the Platform SDK information on Cloaking on MSDN.

Summary

This chapter has described how to build secure serviced components within an Enterprise Services application. You have also seen how to configure an ASP.NET Web-based client application that calls serviced components. To summarize:

- Use server activated Enterprise Services applications for increased security. Additional process hops raise security.
- Use least-privileged, local accounts to run server applications.
- Use Packet Privacy level authentication (which must be configured at the server and client) if you need to secure the data sent to and from a serviced component across a network from a client application.
- Enable component-level access checks for a meaningful role-based security implementation.

- Use Windows authentication and enable impersonation in an ASP.NET Web application prior to calling a component within an Enterprise Services application that relies on role-based security.
- Use secured gateway classes as entry points into Enterprise Service applications.

  By reducing the number of gateway classes that provide entry points for clients into your Enterprise Service applications, you reduce the number of classes that need to have roles assigned. Other internal helper classes should have role-based checks enabled but should have no roles assigned to them. This means that external clients will not be able to call them directly, while gateway classes in the same application will have direct access.

- Call **IsSecurityEnabled** immediately prior to checking role membership programmatically.
- Avoid impersonation in the middle tier because this prevents the effective use of database connection pooling and dramatically reduces the scalability of your application.
- Add Windows groups to Enterprise Services (COM+) roles for increased flexibility and easier administration

Web Services Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
- Microsoft® ASP.NET
- Microsoft® Internet Information Server
- Microsoft® Web Services Development Kit

See the Landing Page for the starting point and complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter focuses on platform-level security for Web services using the underlying features of IIS and ASP.NET. For message-level security, Microsoft is developing the Web Services Development Kit, which allows you to build security solutions that conform to the WS-Security specification, part of the Global XML Web Services Architecture (GXA) initiative. (29 printed pages)

Contents

This chapter describes how to develop and apply authentication, authorization, and secure communication techniques to secure ASP.NET Web services and Web service messages. It describes security from the Web service perspective and shows you how to authenticate and authorize callers and how to flow security context through a Web service. It also explains, from a client-side perspective, how to call Web services with credentials and certificates to support server-side authentication.

Web Service Security Model

Web service security can be applied at three levels:

- Platform/transport-level (point-to-point) security
- Application-level (custom) security
- Message-level (end-to-end) security

Each approach has different strengths and weaknesses, and these are elaborated upon below. The choice of approach is largely dependent upon the characteristics of the architecture and platforms involved in the message exchange.

Note This chapter focuses on platform- and application-level security. Message-level security is addressed by the Global XML Web Services Architecture (GXA) initiative and specifically the WS-Security specification. At the time of writing, Microsoft has just released a technology preview version of the Web Services Development Kit. This allows you to develop message-level security solutions that conform to the WS-Security specification.

Platform/Transport Level (Point-to-Point) Security

The transport channel between two endpoints (Web service client and Web service) can be used to provide point-to-point security. This is illustrated in Figure 10.1.

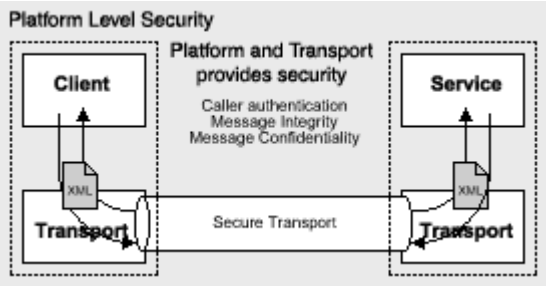

Figure 10.1. Platform/transport-level security

When you use platform security, which assumes a tightly-coupled Microsoft® Windows® operating system environment, for example, on corporate intranets:

- The Web server (IIS) provides Basic, Digest, Integrated, and Certificate authentication.
- The ASP.NET Web service inherits some of the ASP.NET authentication and authorization features.
- SSL and/or IPSec may be used to provide message integrity and confidentiality.

When to use

The transport-level security model is simple, well understood, and adequate for many (primarily intranet-based) scenarios, in which the transport mechanisms and endpoint configuration can be tightly controlled.

The main issues with transport-level security are:

- Security becomes tightly coupled to, and dependant upon, the underlying platform, transport mechanism, and security service provider (NTLM, Kerberos, and so on).
- Security is applied on a point to point basis, with no provision for multiple hops and routing through intermediate application nodes.

Application Level Security

With this approach, the application takes over security and uses custom security features. For example:

- An application can use a custom SOAP header to pass user credentials to authenticate the user with each Web service request. A common approach is to pass a ticket (or user name or license) in the SOAP header.
- The application has the flexibility to generate its own **IPrincipal** object that contains roles. This might be a custom class or the **GenericPrincipal** class provided by the .NET Framework.
- The application can selectively encrypt what it needs to, although this requires secure key storage and developers must have knowledge of the relevant cryptography APIs.

  An alternative technique is to use SSL to provide confidentiality and integrity and combine it with custom SOAP headers to perform authentication.

When to use

Use this approach when:

- You want to take advantage of an existing database schema of users and roles that is used within an existing application.
- You want to encrypt parts of a message, rather than the entire data stream.

Message Level (End-to-End) Security

This represents the most flexible and powerful approach and is the one used by the GXA initiative, specifically within the WS-Security specification. Message-level security is illustrated in Figure 10.2.

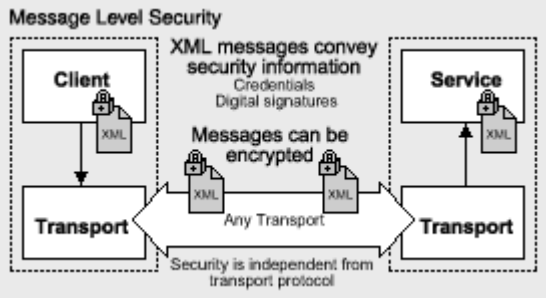

Figure 10.2. Message-level security

WS-Security specifications describe enhancements to SOAP messaging that provide message integrity, message confidentiality, and single message authentication.

- Authentication is provided by security tokens, which flow in SOAP headers. No specific type of token is required by WS-Security. The security tokens may include Kerberos tickets, X.509 certificates, or a custom binary token.
- Secure communication is provided by digital signatures to ensure message integrity and XML encryption for message confidentiality.

When to use

WS-Security can be used to construct a framework for exchanging secure messages in a heterogeneous Web services environment. It is ideally suited to heterogeneous environments and scenarios where you are not in direct control of the configuration of both endpoints and intermediate application nodes.

Message-level security:

- Can be independent from the underlying transport
- Enables a heterogeneous security architecture
- Provides end-to-end security and accommodates message routing through intermediate application nodes
- Supports multiple encryption technologies
- Supports non-repudiation

The Web Services Development Kit

The Web Services Development Kit provides the necessary APIs to manage security in addition to other services such as routing and message-level referrals. This toolkit conforms to the latest Web service standards such as WS-Security and as a result enables interoperability with other vendors who follow the same specifications.

More information

- For the latest news about the Web Services Development Kit and WS-Security specifications, see the XML Web Services Developer Center page.
- For more information about the WS-Specification, see the WS-Security Specification Index page.
- For discussions on this topic, refer to the GXA Interoperability Newsgroup.

Platform / Transport Security Architecture

- The ASP.NET Web services platform security architecture is shown in Figure 10.3.

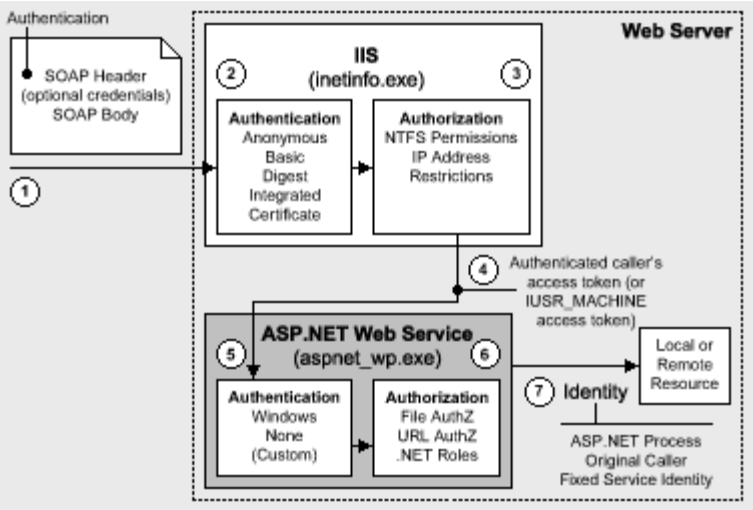

Figure 10.3. Web services security architecture

Figure 10.3 illustrates the authentication and authorization mechanisms provided by ASP.NET Web services. When a client calls a Web service, the following sequence of authentication and authorization events occurs:

1. The SOAP request is received from the network. This may or may not contain authentication credentials depending upon the type of authentication being used.

2. IIS optionally authenticates the caller by using Basic, Digest, Integrated (NTLM or Kerberos), or Certificate authentication. In heterogeneous environments where IIS (Windows) authentication is not possible, IIS is configured for anonymous authentication. In this scenario, the client may be authenticated by using message-level attributes such as tickets passed in the SOAP header.

3. IIS can also be configured to accept requests only from client computers with specific IP addresses.

4. IIS passes the authenticated caller's Windows access token to ASP.NET (this may be the anonymous Internet user's access token, if the Web service is configured for anonymous authentication).

5. ASP.NET authenticates the caller. If ASP.NET is configured for Windows authentication, no additional authentication occurs at this point; IIS authenticates the caller.

   If a non-Windows authentication method is being used, the ASP.NET authentication mode is set to None to allow custom authentication.

   Note Forms and Passport authentication are not currently supported for Web services.

6. ASP.NET authorizes access to the requested Web service (.asmx file) by using URL authorization and File authorization, which uses NTFS permissions associated with the .asmx file to determine whether or not access should be granted to the authenticated caller.

   Note File authorization is only supported for Windows authentication.

   For fine-grained authorization, .NET roles can also be used (either declaratively or programmatically) to ensure that the caller is authorized to access the requested Web method.

7. Code within the Web service may access local and/or remote resources by using a particular identity. By default, ASP.NET Web services perform no impersonation and, as a result, the configured ASP.NET process account provides the identity. Alternate options include the original caller's identity, or a configured service identity.

Gatekeepers

The gatekeepers within an ASP.NET Web service are:

- IIS
  - If IIS anonymous authentication is disabled IIS only allows requests from authenticated users.
  - IP Address Restrictions

    IIS can be configured to only allow requests from computers with specific IP addresses.
- ASP.NET
  - The File authorization HTTP Module (for Windows authentication only)
  - The URL authorization HTTP Module
- Principal Permission Demands and Explicit Role Checks

More information

- For more information about the gatekeepers, see Gatekeepers in Chapter 8, "ASP.NET Security."
- For more information about configuring security, see "Configuring Security" later in this chapter.

Authentication and Authorization Strategies

This section explains which authorization options (configurable and programmatic) are available for a set of commonly used authentication schemes.

The following authentication schemes are summarized here:

- Windows authentication with impersonation
- Windows authentication without impersonation
- Windows authentication using a fixed identity

Windows Authentication with Impersonation

The following configuration elements show you how to enable Windows (IIS) authentication and impersonation declaratively in Web.config or Machine.config.

Note You should configure authentication on a per-Web service basis in each Web service's Web.config file.

```
<authentication mode="Windows" />

<identity impersonate="true" />
```

With this configuration, your Web service code impersonates the IIS-authenticated caller. To impersonate the original caller, you must turn off anonymous access in IIS. With anonymous access, the Web service code impersonates the anonymous Internet user account (which by default is IUSR_MACHINE).

Configurable security

When you use Windows authentication together with impersonation, the following authorization options are available to you:

- **Windows Access Control Lists** (**ACLs)**
  - **Web service (.asmx) file**. File authorization performs access checks for requested ASP.NET resources (which include the .asmx Web service file) using the original caller's security context. The original caller must be granted at least read access to the .asmx file.

- **Resources accessed by your Web service**. Windows ACLs on resources accessed by your Web service (files, folders, registry keys, Active Directory® directory service objects and so on) must include an Access Control Entry (ACE) that grants read access to the original caller (because the Web service thread used for resource access is impersonating the caller).

- **URL Authorization.** This is configured in Machine.config and/or Web.config. With Windows authentication, user names take the form DomainName\UserName and roles map one-to-one with Windows groups.

```
<authorization>
  <deny user="DomainName\UserName" />
  <allow roles="DomainName\WindowsGroup" />
</authorization>
```

Programmatic security

Programmatic security refers to security checks located within your Web service code. The following programmatic security options are available when you use Windows authentication and impersonation:

- **Principal Permission Demands**
  - Imperative (in-line within a method's code)

```
PrincipalPermission permCheck = new PrincipalPermission(
                                        null,
@"DomainName\WindowsGroup");
permCheck.Demand();
```

  - Declarative (these attributes can precede Web methods or Web classes)

```
// Demand that the caller is a member of a specific role (for
// Windows
// authentication this is the same as a Windows group)
[PrincipalPermission(SecurityAction.Demand,
                     Role=@"DomainName\WindowsGroup")]
// Demand that the caller is a specific user
[PrincipalPermission(SecurityAction.Demand,
                     Name=@"DomainName\UserName")]
```

- **Explicit Role Checks**. You can perform role checking using the **IPrincipal** interface.

```
IPrincipal.IsInRole(@"DomainName\WindowsGroup");
```

When to use

Use Windows authentication and impersonation when:

- The clients of the Web service can be identified by using Windows accounts, which can be authenticated by the server.
- You need to flow the original caller's security context through the Web service and onto the next tier. For example, a set of serviced components that use Enterprise Services (COM+) roles, or onto a data tier that requires fine-grained (per-user) authorization.
- You need to flow the original caller's security context to the downstream tiers to support operating system-level auditing.

Important Using impersonation can reduce scalability, because it impacts database connection pooling. As an alternative approach, consider using the trusted subsystem model where the Web service authorizes callers and then uses a fixed identity for database access. You can flow the caller's identity at the application level; for example, by using stored procedure parameters.

More information

- For more information about Windows authentication and impersonation, see Chapter 8, ASP.NET Security.
- For more information about URL authorization, see URL Authorization Notes in Chapter 8, "ASP.NET Security."

Windows Authentication without Impersonation

The following configuration elements show how you enable Windows (IIS) authentication with no impersonation declaratively in Web.config.

```
<authentication mode="Windows" />

<!-- The following setting is equivalent to having no identity element -->

<identity impersonate="false" />
```

Configurable security

When you use Windows authentication without impersonation, the following authorization options are available to you:

- **Windows ACLs**
  - **Web Service (.asmx) file**. File authorization performs access checks for requested ASP.NET resources (which include the .asmx Web service file) using the original caller. Impersonation is not required.
  - **Resources accessed by your application**. Windows ACLs on resources accessed by your application (files, folders, registry keys, Active Directory objects) must include an ACE that grants read access to the ASP.NET process identity (the default identity used by the Web service thread when accessing resources).
- **URL Authorization**

  This is configured in Machine.config and Web.config. With Windows authentication, user names take the form DomainName\UserName and roles map one-to-one with Windows groups.

```
<authorization>

  <deny user="DomainName\UserName" />

  <allow roles="DomainName\WindowsGroup" />
```

```
</authorization>
```

Programmatic security

Programmatic security refers to security checks located within your Web service code. The following programmatic security options are available when you use Windows authentication without impersonation:

- **Principal Permission Demands**
  - Imperative

```
PrincipalPermission permCheck = new PrincipalPermission(
                                        null,
@"DomainName\WindowsGroup");
permCheck.Demand();
```

  - Declarative

```
// Demand that the caller is a member of a specific role (for
// Windows
// authentication this is the same as a Windows group)
[PrincipalPermission(SecurityAction.Demand,
                     Role=@"DomainName\WindowsGroup")]
// Demand that the caller is a specific user
[PrincipalPermission(SecurityAction.Demand,
                     Name=@"DomainName\UserName")]
```

- **Explicit Role Checks**. You can perform role checking using the **IPrincipal** interface.

```
IPrincipal.IsInRole(@"DomainName\WindowsGroup");
```

When to use

Use Windows authentication without impersonation when:

- The clients of the Web service can be identified by using Windows accounts, which can be authenticated by the server.
- You want to use the trusted subsystem model and authorize clients within the Web service and then use a fixed identity to access downstream resources (for example, databases) in order to support connection pooling.

More information

- For more information about Windows authentication and impersonation, see Chapter 8, ASP.NET Security.

- For more information about URL authorization, see URL Authorization Notes in Chapter 8, "ASP.NET Security."

Windows Authentication Using a Fixed Identity

The <**identity**> element within Web.config supports optional user name and password attributes which allows you to configure a specific fixed identity for your Web service to impersonate. This is shown in the following configuration file fragment.

```
<identity impersonate="true" userName="DomainName\UserName"
                             password="ClearTextPassword" />
```

When to use

This approach is not recommended in secure environments for two reasons:

- User names and passwords should not be stored in plain text in configuration files.
- On Windows 2000, this approach forces you to grant the ASP.NET process account the "Act as part of the operating system" privilege. This reduces the security of your Web service and increases the threat should an attacker compromise the Web service process (Aspnet_wp.exe).

More information

- For more information about Windows authentication and impersonation, see Chapter 8, ASP.NET Security.
- For more information about URL authorization, see URL Authorization Notes in Chapter 8, "ASP.NET Security."

Configuring Security

This section shows you the practical steps required to configure security for an ASP.NET Web service. These are summarized in Figure 10.4.

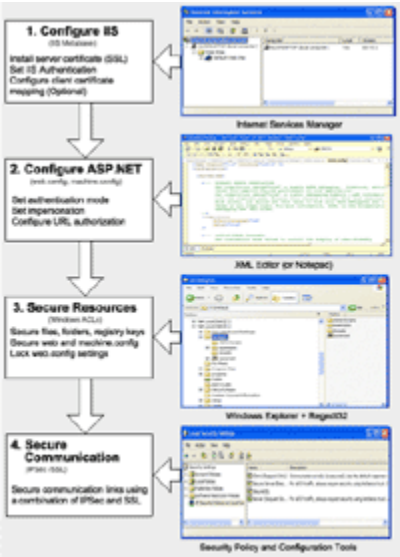

Figure 10.4. Configuring ASP.NET Web service security

Configure IIS Settings

For detailed information about how to configure IIS security settings, see Configuring Security in Chapter 8, "ASP.NET Security," because the information is also applicable to ASP.NET Web services.

Configure ASP.NET Settings

Application-level configuration settings are maintained in Web.config files, which are located in your Web service's virtual root directory. Configure the following settings:

1. **Configure Authentication**. This should be set on a per-Web service basis (not in Machine.config) in the Web.config file located in the Web service's virtual root directory.

```
2.      <authentication mode="Windows|None" />
```

   Note Web services do not currently support Passport or Forms authentication. For custom and message-level authentication, set the mode to None.

3. **Configure Impersonation and Authorization**. For detailed information, see Configuring Security in Chapter 8, "ASP.NET Security."

More information

For more information about URL authorization, see URL Authorization Notes in Chapter 8, "ASP.NET Security."

Secure Resources

You should use the same techniques to secure Web resources as presented in Chapter 8, ASP.NET Security. In addition, however, for Web services consider removing the HTTP-GET and HTTP-POST protocol from Machine.config on production servers.

Disable HTTP-GET, HTTP-POST

By default, clients can communicate with ASP.NET Web services, using three protocols: HTTP-GET, HTTP-POST, and SOAP over HTTP. You should disable support for both the HTTP-GET and HTTP-POST protocols at the machine level on production machines that do not require them. This is to avoid a potential security breach that could allow a malicious Web page to access an internal Web service running behind a firewall.

Note Disabling these protocols means that a new client will not be able to test an XML Web service using the **Invoke** button on the Web service test page. Instead, you must create a test client program by adding a reference to the Web service using Microsoft Visual Studio® .NET development system. You may want to leave these protocols enabled on development computers to allow developers to use the test page.

To disable the HTTP-GET and HTTP-POST protocols for an entire computer

1. Edit Machine.config.
2. Comment out the lines within the <**webServices**> element that add support for HTTP-GET and HTTP-POST. After doing so, Machine.config should appear as follows.

```
3.      <webServices>
4.          <protocols>
5.            <add name="HttpSoap"/>
6.               <!-- <add name="HttpPost"/> -->
7.               <!-- <add name="HttpGet"/>  -->
8.            <add name="Documentation"/>
9.          </protocols>
```

```
10.     </webServices>
```

11. Save Machine.config.

Note For special cases where you have Web service clients that communicate with a Web service using either HTTP-GET or HTTP-POST, you can add support for those protocols in the application's Web.config file, by creating a <**webServices**> and adding support for these protocols with the <**protocol**> and <**add**> elements, as shown earlier.

More information

For detailed Information about securing resources, see Secure Resources within Chapter 8, "ASP.NET Security."

Secure Communication

Use a combination of SSL and IPSec to secure communication links.

More information

- For information about calling a Web service using SSL, see How To: Call a Web Service Using SSL in the Reference section of this guide.
- For information about using IPSec between two computers, see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Passing Credentials for Authentication to Web Services

When you call a Web service, you do so by using a Web service proxy; a local object that exposes the same set of methods as the target Web service.

You can generate a Web service proxy by using the Wsdl.exe command line utility. Alternatively, if you are using Visual Studio .NET you can generate the proxy by adding a Web reference to the project.

Note If the Web service for which you want to generate a proxy is configured to require client certificates, you must temporarily switch off that requirement while you add the reference, or an error occurs. After you add the reference, you must remember to reconfigure the service to require certificates.
An alternate approach would be to keep an offline Web Services Description Language (WSDL) file available to consumer applications. You must remember to update this if your Web service interface changes.

Specifying Client Credentials for Windows Authentication

If you are using Windows authentication, you must specify the credentials to be used for authentication using the **Credentials** property of the Web service proxy. If you do not explicitly set this property, the Web service is called without any credentials. If Windows authentication is required, this will result in an HTTP status 401, access denied response.

Using DefaultCredentials

Client credentials do not flow implicitly. The Web service consumer must set the credentials and authentication details on the proxy. To flow the security context of the client's Windows security context (either from an impersonating thread token or process token) to a Web service you can set the **Credentials** property of the Web service proxy to **CredentialCache**. **DefaultCredentials** as shown below.

```
proxy.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

Consider the following points before you use this approach:

- This flows the client credentials only when you use NTLM, Kerberos, or Negotiate authentication.
- If a client-side application (for example, a Windows Forms application) calls the Web service, the credentials are obtained from the user's interactive logon session.

- Server-side applications—such as ASP.NET Web applications—use the process identity unless impersonation is configured. In that case the impersonated caller's identity is used.

Using specific credentials

To use a specific set of credentials for authentication when you call a Web service, use the following code.

```
CredentialCache cache = new CredentialCache();

cache.Add( new Uri(proxy.Url), // Web service URL

           "Negotiate",        // Kerberos or NTLM

           new NetworkCredential("username", "password", "domainname") );

proxy.Credentials = cache;
```

In the above example, the requested Negotiate authentication type results in either Kerberos or NTLM authentication.

Always request a specific authentication type

You should always request a specific authentication type as illustrated above. Avoid direct use of the **NetworkCredential** class as shown in the following code.

```
proxy.Credentials = new

                      NetworkCredential("username", "password",
"domainname");
```

This should be avoided in production code because you have no control over the authentication mechanism used by the Web service and as a result you have no control over how the credentials are used.

For example, you may expect a Kerberos or NTLM authentication challenge from the server but instead you may receive a Basic challenge. In this case, the supplied user name and password will be sent to the server in clear text form.

Set the PreAuthenticate property

The proxy's **PreAuthenticate** property can be set to true or false. Set it to true to supply specific authentication credentials to cause a **WWW-authenticate** HTTP header to be passed with the Web request. This saves the Web server denying access on the request, and performing authentication on the subsequent retry request.

Note Pre-authentication only applies after the Web service successfully authenticates the first time. Pre-authentication has no impact on the first Web request.

```
private void ConfigureProxy( WebClientProtocol proxy,

                             string domain, string username,

                             string password )

{

  // To improve performance, force pre-authentication

  proxy.PreAuthenticate = true;

  // Set the credentials
```

```
  CredentialCache cache = new CredentialCache();

  cache.Add( new Uri(proxy.Url),

             "Negotiate",

             new NetworkCredential(username, password, domain) );

  proxy.Credentials = cache;

  proxy.ConnectionGroupName = username;

}
```

Using the ConnectionGroupName property

Notice that the above code sets the **ConnectionGroupName** property of the Web service proxy. This is only required if the security context used to connect to the Web service varies from one request to the next as described below.

If you have an ASP.NET Web application that connects to a Web service and flows the security context of the original caller (by using **DefaultCredentials** or by setting explicit credentials, as shown above), you should set the **ConnectionGroupName** property of the Web service proxy within the Web application. This is to prevent a new, unauthenticated client from reusing an old, authenticated TCP connection to the Web service that is associated with a previous client's authentication credentials. Connection reuse can occur as a result of HTTP KeepAlives and authentication persistence which is enabled for performance reasons within IIS.

Set the **ConnectionGroupName** property to an identifier (such as the caller's user name) that distinguishes one caller from the next as shown in the previous code fragment.

Note If the original caller's security context does not flow through the Web application and onto the Web service, and instead the Web application connects to the Web service using a fixed identity (such as the Web application's ASP.NET process identity), you do not need to set the **ConnectionGroupName** property. In this scenario, the connection security context remains constant from one caller to the next.

Calling Web Services from Non-Windows Clients

There are a number of authentication approaches that work for cross-browser scenarios. These include:

- **Certificate Authentication**. This uses cross-platform X.509 certificates.
- **Basic Authentication**. For an example of how to use Basic authentication against a custom data store (without requiring Active Directory), see Web Services Security - HTTP Basic Authentication without Active Directory.
- **GXA Message Level Approaches**. Use the Web Services Development Toolkit to implement GXA (WS-Security) solutions.
- **Custom Approaches**. For example, flow credentials in SOAP headers.

Proxy Server Authentication

Proxy server authentication is not supported by the Visual Studio .NET **Add Web Reference** dialog box (although it will be supported with the next version of Visual Studio .NET). As a result you might receive an HTTP status 407: "Proxy Authentication Required" response when you attempt to add a Web reference.

Note You may not see this error when you view the .asmx file from a browser, because the browser automatically sends credentials.

To work around this issue, you can use the Wsdl.exe command line utility (instead of the **Add Web Reference** dialog) as shown below.

```
wsdl.exe /proxy:http://<YourProxy> /pu:<YourName> /pp:<YourPassword>
/pd:<YourDomain> http://www.YouWebServer.com/YourWebService/YourService.asmx
```

If you need to programmatically set the proxy server authentication information, use the following code.

```
YourWebServiceProxy.Proxy.Credentials = CredentialsCache.DefaultCredentials;
```

Flowing the Original Caller

This section describes how you can flow the original caller's security context through an ASP.NET Web application and onto a Web service located on a remote application server. You may need to do this in order to support per-user authorization within the Web service or within subsequent downstream subsystems (for example, databases, where you want to authorize original callers to individual database objects).

In Figure 10.5, the security context of the original caller (Alice) flows through the front-end Web server that hosts an ASP.NET Web application, onto the remote object, hosted by ASP.NET on a remote application server and finally through to a backend database server.

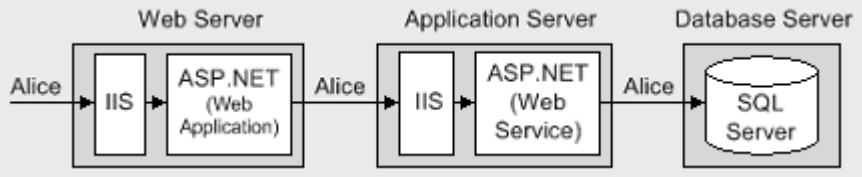

Figure 10.5. Flowing the original caller's security context

In order to flow credentials to a Web service, the Web service client (the ASP.NET Web application in this scenario) must configure the Web service proxy and explicitly set the proxy's **Credentials** property, as described in "Passing Credentials for Authentication to Web Services" earlier in this chapter.

There are two ways to flow the caller's context.

- **Pass default credentials and use Kerberos authentication (and delegation)**. This approach requires that you impersonate within the ASP.NET Web application and configure the remote object proxy with **DefaultCredentials** obtained from the impersonated caller's security context.
- **Pass explicit credentials and use Basic or Forms authentication**. This approach does not require impersonation within the ASP.NET Web application. Instead, you programmatically configure the Web service proxy with explicit credentials obtained from either server variables (with Basic authentication) or HTML form fields (with Forms authentication) that are available to the Web application. With Basic or Forms authentication, the user name and password are available to the server in clear text.

Default Credentials with Kerberos Delegation

To use Kerberos delegation, all computers (servers and clients) must be running Windows 2000 or later. Additionally, client accounts that are to be delegated must be stored in Active Directory and must not be marked as "Sensitive and cannot be delegated."

The following tables show the configuration steps required on the Web server, and application server.

Configuring the Web server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web application's virtual root directory Enable Windows Integrated	 Kerberos authentication will be negotiated assuming clients and server are

Authentication for the Web application's virtual root	running Windows 2000 or later. **Note**: If you are using Internet Explorer 6 on Windows 2000, it defaults to NTLM authentication instead of the required Kerberos authentication. To enable Kerberos delegation, see article Q299838, Unable to Negotiate Kerberos Authentication after upgrading to Internet Explorer 6, in the Microsoft Knowledge Base.
Configure ASP.NET	
Step	**More Information**
Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your Web application's virtual directory Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure your ASP.NET Web application for impersonation	Edit Web.config in your Web application's virtual directory Set the <**identity**> element to: `<identity impersonate="true" />`
Configure the Web Service Proxy	
Step	**More Information**
Set the credentials property of the Web service proxy to **DefaultCredentials**.	See Using DefaultCredentials earlier in this chapter for a code sample.

Configuring the remote application server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web service's virtual root directory Enable Windows Integrated Authentication for the Web application's virtual root	
Configure ASP.NET (Web Service Host)	
Step	**More Information**
Configure ASP.NET to use Windows authentication	Edit Web.config in the Web service's virtual directory. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure ASP.NET for impersonation	Edit Web.config in the Web service's virtual directory. Set the <**identity**> element to: `<identity impersonate="true" />` **Note**: This step is only required if you want to flow the original caller's security context through the Web service and onto the next downstream, subsystem (for example, a database). With impersonation enabled here, resource access (local and remote) uses the impersonated original caller's security context.

	If your requirement is simply to allow per-user authorization checks in the Web service, you do not need to impersonate here.

More information

For more information about configuring Kerberos delegation, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.

Explicit Credentials with Basic or Forms Authentication

As an alternative to Kerberos delegation, you can use Basic or Forms authentication at the Web application to capture the client's credentials and then use Basic (or Integrated Windows) authentication to the Web service.

With this approach, the client's clear-text credentials are available to the Web application. These can be passed to the Web service through the Web service proxy. For this, you must write code in the Web application to retrieve the client's credentials and configure the proxy.

Basic authentication

With Basic authentication, the original caller's credentials are available to the Web application in server variables. The following code shows how to retrieve them and configure the Web service proxy.

```
// Retrieve client's credentials (available with Basic

// authentication)

string pwd = Request.ServerVariables["AUTH_PASSWORD"];

string uid = Request.ServerVariables["AUTH_USER"];

// Associate the credentials with the Web service proxy

// To improve performance, force preauthentication

proxy.PreAuthenticate = true;

// Set the credentials

CredentialCache cache = new CredentialCache();

cache.Add( new Uri(proxy.Url),

           "Basic",

           new NetworkCredential(uid, pwd, domain) );

proxy.Credentials = cache;
```

Forms authentication

With Forms authentication, the original caller's credentials are available to the Web application in form fields (rather than server variables). In this case, use the following code.

```
// Retrieve client's credentials from the logon form

string pwd = txtPassword.Text;
```

```
string uid = txtUid.Text;

// Associate the credentials with the Web service proxy

// To improve performance, force preauthentication

proxy.PreAuthenticate = true;

// Set the credentials

CredentialCache cache = new CredentialCache();

cache.Add( new Uri(proxy.Url),

           "Basic",

           new NetworkCredential(uid, pwd, domain) );

proxy.Credentials = cache;
```

The following tables show the configuration steps required on the Web server, and application server.

Configuring the Web server

Configure IIS	
Step	**More Information**
To use Basic authentication, disable Anonymous access for your Web application's virtual root directory and select Basic authentication - or - To use Forms authentication, enable anonymous access	Both Basic and Forms authentication should be used in conjunction with SSL to protect the clear text credentials sent over the network. If you use Basic authentication, SSL should be used for all pages (not just the initial logon page), as Basic credentials are transmitted with every request. Similarly, SSL should be used for all pages if you use Forms authentication, to protect the clear text credentials on the initial log on and to protect the authentication ticket passed on subsequent requests.
Configure ASP.NET	
Step	**More Information**

If you use Basic authentication, configure your ASP.NET Web application to use Windows authentication
 	
- or -
 	
If you use Forms authentication, configure your ASP.NET Web application to use Forms authentication
	Edit Web.config in your Web application's virtual directory
	Set the <**authentication**> element to:
	`<authentication mode="Windows" />`
	
	- or -
	
	Edit Web.config in your Web application's virtual directory
	Set the <**authentication**> element to:
	`<authentication mode="Forms" />`

Disable impersonation within the ASP.NET Web application	Edit Web.config in your Web application's virtual directory. Set the <**identity**> element to:

	`<identity impersonate="false" />` **Note:** This is equivalent to having no <**identity**> element. Impersonation is not required, as the user's credentials will be passed explicitly to the Web service through the proxy.
Configure the Web Service Proxy	
Step	**More Information**
Write code to capture and explicitly set the credentials on the Web Service proxy	Refer to the code fragments shown earlier in the Basic Authentication and Forms Authentication sections.

Configuring the application server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your application's virtual root directory Enable Basic authentication	**Note:** Basic authentication at the (Web service) application server, allows the Web service to flow the original caller's security context to the database (as the caller's user name and password are available in clear text and can be used to respond to network authentication challenges from the database server). If you don't need to flow the original caller's security context beyond the Web service, consider configuring IIS at the application server to use Windows Integrated authentication, as this provides tighter security—credentials are not passed across the network and are not available to the Web service.
Configure ASP.NET (Web Service)	
Step	**More Information**
Configure ASP.NET to use Windows authentication	Edit Web.config in the Web service's virtual directory. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure your ASP.NET Web service for impersonation	Edit Web.config in the Web service's virtual directory. Set the <**identity**> element to: `<identity impersonate="true" />` **Note:** This step is only required if you want to flow the original caller's security context through the Web service and onto the next downstream, subsystem (for example, a database). With impersonation enabled here, resource access (local and remote) uses the impersonated original caller's security context. If your requirement is simply to allow per-user authorization checks in the Web service, you do not need to impersonate here.

Trusted Subsystem

The trusted subsystem model provides an alternative (and simpler to implement) approach to flowing the original caller's security context. In this model a trust boundary exists between the Web service and Web application. The Web service trusts the Web application to properly authenticate and authorize callers, prior to letting requests proceed to the Web service. No authentication of the original callers occurs at the Web service. The Web service authenticates the fixed trusted identity used by the Web application to communicate with the Web service. In most cases, this is the process identity of the ASP.NET worker process.

The trusted subsystem model is shown in Figure 10.6.

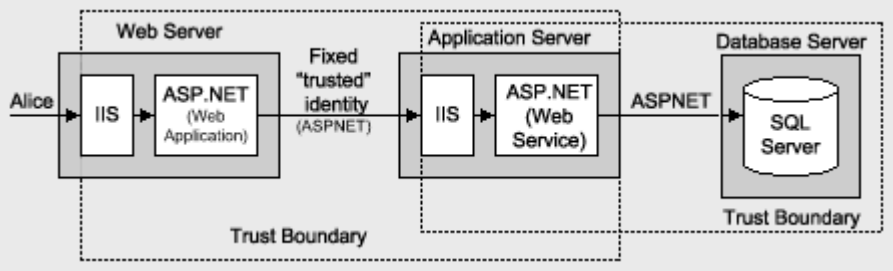

Figure 10.6. The trusted subsystem model

Flowing the Caller's Identity

If you use the trusted subsystem model, you may still need to flow the original caller's identity (name, not security context), for example, for auditing purposes at the database.

You can flow the identity at the application level by using method and stored procedure parameters and trusted query parameters (as shown in the following example) can be used to retrieve user-specific data from the database.

```
SELECT x,y,z FROM SomeTable WHERE UserName = "Alice"
```

Configuration Steps

The following tables show the configuration steps required on the Web server, and application server.

Configuring the web server

Configure IIS	
Step	**More Information**
Configure IIS authentication	The Web application can use any form of authentication to authenticate the original callers.
Configure ASP.NET	
Step	**More Information**

Configure authentication and make sure impersonation is disabled	Edit Web.config in your Web application's virtual directory.
	Set the <**authentication**> element to "Windows", "Forms" or "Passport."
	`<authentication mode="Windows|Forms|Passport" />`
	Set the <**identity**> element to:
	`<identity impersonate="false" />`
	(or remove the <**identity**> element)

Reset the password of the ASPNET	For more information about how to access network resources (including Web

account used to run ASP.NET OR create a least privileged domain account to run ASP.NET and specify account details on the <**processModel**> element within Web.config	services) from ASP.NET and about choosing and configuring a process account for ASP.NET, see Accessing Network Resources and Process Identity for ASP.NET in Chapter 8, "ASP.NET Security."
Configure Web Service Proxy	
Step	**More Information**
Configure the Web service proxy to use default credentials for all calls to the Web service	Use the following line of code: proxy.Credentials = DefaultCredentials;

Configuring the application server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web service's virtual root directory Enable Windows Integrated authentication	
Configure ASP.NET	
Step	**More Information**
Configure ASP.NET to use Windows authentication	Edit Web.config in the Web service's virtual directory. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Disable impersonation	Edit Web.config in the application's virtual directory. Set the <**identity**> element to: `<identity impersonate="false" />`

Accessing System Resources

For details about accessing system resources (for example the event log and the registry) from ASP.NET Web services, see Accessing System Resources in Chapter 8, "ASP.NET Security." The approaches and restrictions discussed in Chapter 8 also apply to ASP.NET Web services.

Accessing Network Resources

When you access network resources from a Web service, you need to consider the identity that is used to respond to network authentication challenges from the remote computer. You have three options:

- The process identity (determined by the account used to run the ASP.NET worker process)
- A service identity (for example, one created by calling **LogonUser**)
- The original caller identity (with the Web service configured for impersonation)

For details about the relative merits of each of these approaches, together with configuration details, see Accessing Network Resources in Chapter 8, "ASP.NET Security."

Accessing COM Objects

The **AspCompat** directive (used by Web applications when they call apartment threaded COM objects) is not available to Web services. This means that when you call apartment model objects from Web services, a thread switch occurs. This results in a slight performance hit, and if your Web service is impersonating, your impersonation token will be lost when calling the COM component. This typically results in an Access Denied exception.

More Information

- For more information about access denied exceptions when calling apartment threaded COM objects, see article Q325791,PRB: Access Denied Error Message Occurs When Impersonating in ASP.NET and Calling STA COM Components, in the Microsoft Knowledge Base.
- For more information about accessing COM objects and using the AspCompat attribute, see Accessing COM Objects within Chapter 8, "ASP.NET Security."
- For more information about calling apartment threaded COM objects from Web services, see article Q303375, INFO: XML Web Services and Apartment Objects, in the Microsoft Knowledge Base.

Using Client Certificates with Web Services

This section describes techniques for using X.509 client certificates for Web service authentication.

You can use client certificate authentication within a Web service to authenticate:

- Other Web services
- Applications that communicate directly with the Web service (for example, server-based or client-side desktop applications)

Authenticating Web Browser Clients with Certificates

A Web service cannot use client certificates to authenticate callers if they interact with an intermediate Web application, because it is not possible to forward the original caller's certificate through the Web application and onto the Web service. While the Web application can authenticate its clients with certificates, the same certificates cannot then be used by the Web service for authentication.

The reason that this server-to-server scenario fails is that the Web application does not have access to the client's certificate (specifically its private key) in its certificate store. This problem is illustrated in Figure 10.7.

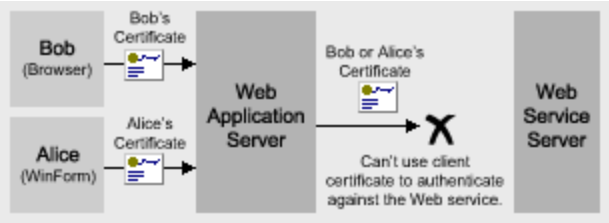

Figure 10.7. Web service client certificate authentication

Using the Trusted Subsystem Model

To address this restriction, and to support certificate authentication at the Web service, you must use a trusted-subsystem model. With this approach, the Web service authenticates the Web application using the Web application's certificate (and not the original caller's certificate). The Web service must trust the Web application to authenticate its users and to perform the necessary authorization to ensure that only authorized callers are able to access the data and functionality exposed by the Web service.

This approach is shown in Figure 10.8.

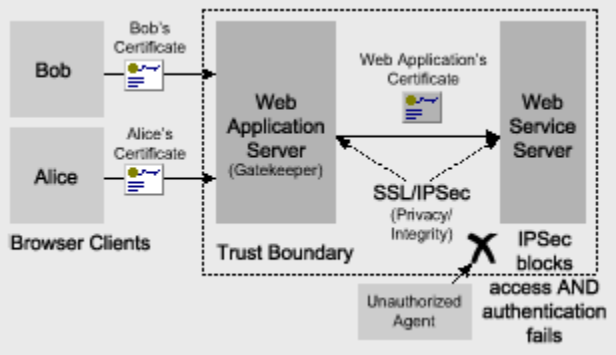

Figure 10.8. The Web service authenticates the trusted Web application

If the authorization logic within the Web service requires multiple roles, the Web application can send different certificates based upon the role membership of the caller. For example, one certificate may be used for members of an Administrators group (who are allowed to update data within a back-end database) and another certificate may be used for all other users (who are authorized only for read operations).

Note In scenarios such as these, a local certificate server (accessible only by the two servers) can be used to manage all the Web application certificates.

In this scenario:

- The Web application authenticates its users by using client certificates.
- The Web application acts as a gatekeeper and authorizes its users and controls access to the Web service.
- The Web application calls the Web service and passes a different certificate that represents the application (or possibly a range of certificates based on the role membership of the caller).
- The Web service authenticates the Web application and it trusts the application to perform the necessary client authorization.
- IPSec is used between the Web application server and Web service server to provide additional access control. Unauthorized access attempts from other computers are prevented by IPSec. Certificate authentication at the Web service server also prevents unauthorized access.

Solution implementation

To use certificate authentication at the Web service in this scenario, use a separate process to call the Web service and pass the certificate. You cannot manipulate the certificates directly from the ASP.NET Web application because it does not have a loaded user profile and associated certificate store. The separate process must be configured to run using an account that has an associated user profile (and certificate store). You have two main options:

- You can use an Enterprise Services server application.
- You can use a Windows service.

Figure 10.9 illustrates this scenario with an Enterprise Services server application.

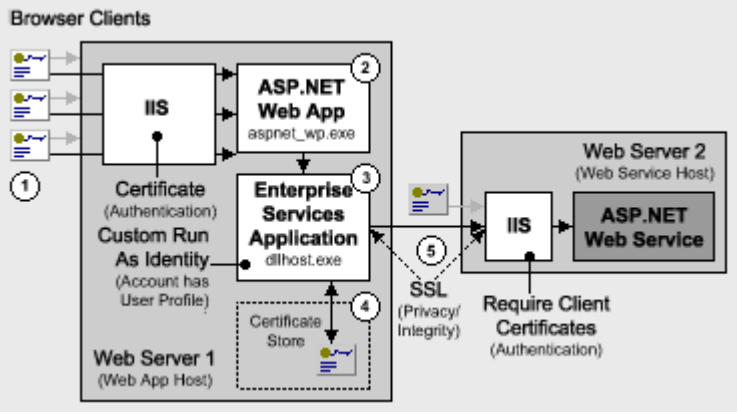

Figure 10.9. Client certificate authentication with Web services

The following summarizes the sequence of events illustrated by Figure 10.9:

1. The original caller is authenticated by the Web application using client certificates.
2. The Web application is the gatekeeper and is responsible for authorizing access to specific areas of functionality (including those that involve interaction with the Web service).
3. The Web application calls a serviced component running in an out-of-process Enterprise Services application.
4. The account used to run the Enterprise Services application has an associated user profile. The component accesses a client certificate from its certificate store, which is used by the Web service to authenticate the Web application.
5. The serviced component calls the Web service, passing the client certificate on each method request. The Web service authenticates the Web application using this certificate and trusts the Web application to correctly authorize original callers.

Why use an additional process?

An additional process is required (rather than using the Aspnet_wp.exe Web process to contact the Web service) due to the fact that a user profile (containing a certificate store) is required.

A Web application that runs using the ASPNET account does not have access to any certificates on the Web server. This is because certificate stores are maintained within user profiles associated with interactive user accounts. User profiles are only created for interactive accounts when you physically log on using the account. The ASPNET account is not intended to be an interactive user account and is configured with the "Deny interactive logon" privilege for added security.

Important Do not reconfigure the ASPNET account to remove this privilege and turn the account into an interactive logon account. Use a separate process with a configured service account to access certificates, as described earlier in this chapter.

More information

- For more information about how to implement this approach, see How To: Call a Web Service Using Client Certificates from ASP.NET in the Reference section of this guide.
- For more information about configuring IPSec, see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.

Secure Communication

Secure communication is concerned with guaranteeing the integrity and confidentiality of Web service messages as they flow from application to application across the network. There are two approaches to this problem: transport-level options and message-level options.

Transport Level Options

Transport-level options include:

- SSL
- IPSec

These options may be appropriate if the following conditions are met:

- You are sending a message directly from your application to a Web service and the message will not be routed through intermediate systems.
- You can control the configuration of both endpoints involved in the message transfer.

Message-Level Options

Message-level approaches can be used to guarantee the confidentiality and integrity of messages as they pass through an arbitrary number of intermediate systems. Messages can be signed to provide integrity. For confidentiality, you can choose between encrypting the entire message or part of a message.

Use a message-level approach if the following conditions are met:

- You are sending a message to a Web service, and the message is likely to be forwarded to other Web services or may be routed through intermediate systems.
- You do not control the configuration of both endpoints. For example, if you are sending messages from one company to another.

More information

- The Web Service Development Toolkit will provide message encryption functionality in accordance with the WS-Security specification.
- For more information about SSL and IPSec, see Chapter 4, Secure Communication.

Summary

This chapter has focused on platform/transport-level (point-to-point) Web service security provided by the underlying services of ASP.NET, IIS, and the operating system. While platform-level security provides secure solutions for tightly-coupled intranet scenarios, it is not suited to heterogeneous scenarios. For this, message-level security provided by the GXA WS-Security specification is required. Use the Web Services Development Kit to build message-level Web service security solutions.

For tightly-coupled Windows domain environments:

- If you want to flow the original caller's identity from an ASP.NET Web application to a remote Web service, the ASP.NET Web application should use Kerberos authentication (with accounts configured for delegation), Basic authentication, or Forms authentication.
  - With Kerberos authentication, enable impersonation with the Web application and configure the **Credentials** property of the Web service proxy using **DefaultCredentials**.
  - With Basic or Forms authentication, capture the caller's credentials and set the **Credentials** property of the Web service proxy by adding a new **CredentialCache** object.

For Web-service to Web-service scenarios:

- Use Basic or Kerberos authentication and set credentials in the client proxy.

- Use an out of process Enterprise Services application or a Windows service to manipulate X.509 certificates from Web applications.
- As far as possible, use system-level authorization checks such as File and URL authorization.
- For granular authorization (for example, at the Web method level) use .NET roles (declaratively and imperatively).
- Authorize non-Windows users by using .NET roles (based on a **GenericPrincipal** object that contains roles).
- Disable HTTP-GET and HTTP-POST protocols on product servers.
- Use transport-level security if you are not worried about passing messages securely through intermediary systems.
- Use transport-level security if SSL performance is acceptable.
- Use WS-Security and the Web Services Development Kit to develop message-level solutions.

.NET Remoting Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and complete overview of Building Secure ASP.NET Applications.

Summary: The .NET Framework provides a remoting infrastructure that allows clients to communicate with objects, hosted in remote application domains and processes, or on remote computers. This chapter shows you how to implement secure .NET Remoting solutions. (27 printed pages)

Contents

The .NET Framework provides a remoting infrastructure that allows clients to communicate with objects, hosted in remote application domains and processes, or on remote computers. This chapter shows you how to implement secure .NET Remoting solutions.

.NET Remoting Architecture

Figure 11.1 shows the basic .NET Remoting architecture when a remote object is hosted within ASP.NET. An ASP.NET host, coupled with the HTTP channel for communication, is the recommended approach if security is the key concern, because it allows the remote object to utilize the underlying security services provided by ASP.NET and IIS.

For more information about the range of possible host and channel types, together with comparison information, see Choosing a Host Process later in this chapter.

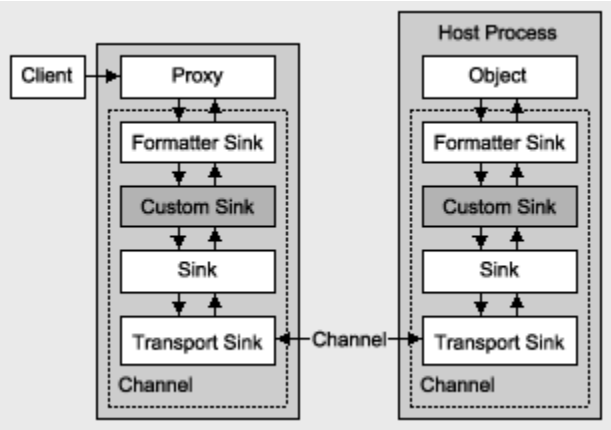

Figure 11.1. The .NET remoting architecture

The client communicates with an in-process proxy object. Credentials for authentication (for example, user names, passwords, certificates, and so on) can be set through the remote object proxy. The method call proceeds through a chain of sinks (you can implement your own custom sinks, for example, to perform data encryption) and onto a transport sink that is responsible for sending the data across the network. At the server side, the call passes through the same pipeline after which the call is dispatched to the object.

Note The term *proxy* used throughout this chapter refers to the client-side, in-process proxy object through which clients communicate with the remote object. Do not confuse this with the term *proxy server*.

Remoting Sinks

.NET Remoting uses transport channels sinks, custom channel sinks, and formatter channel sinks when a client invokes a method call on a remote object.

Transport channel sinks

Transport channel sinks pass method calls across the network between the client and the server. .NET supplies the **HttpChannel** and the **TcpChannel** classes, although the architecture is fully extensible and you can plug in your own custom implementations.

- **HttpChannel**. This channel is designed to be used when you host a remote object in ASP.NET. This channel uses the HTTP protocol to send messages between the client and the server.
- **TcpChannel**. This channel is designed to be used when you host a remote object in a Microsoft® Windows® operating system service or other executable. This channel uses TCP sockets to send messages between the client and the server.
- **Custom channels**. A custom transport channel can use any underlying transport protocol to send messages between the client and server. For example, a custom channel may use named pipes or mail slots.

Comparing Transport Channel Sinks

The following table provides a comparison of the two main transport channel sinks.

Table 11.1. Comparison of TcpChannel and HttpChannel

Feature	TCP Channel	HTTP Channel	Comments
Authentication	No	Yes	The HTTP channel uses the authentication features provided by IIS and ASP.NET, although Passport and Forms authentication is not supported.
Authorization	No	Yes	The HTTP channel supports the authorization features provided by IIS and ASP.NET. These include NTFS permissions, URL authorization and File authorization.
Secure Communication	Yes	Yes	Use IPSec with the TCP channel. Use SSL and/or IPSec with the HTTP channel.

Custom sinks

Custom channels sinks can be used at different locations within the channel sink pipeline to modify the messages sent between the client and the server. A channel sink that provides encryption and decryption is an example of a custom channel sink.

Formatter sinks

Formatter sinks take method calls and serialize them into a stream capable of being sent across the network. .NET supplies two formatter sinks:

- **Binary Formatter**. This uses the **BinaryFormatter** class to package method calls into a serialized binary stream, which is subsequently posted (using an HTTP POST) to send the data to the server. The binary formatter sets the content-type in the HTTP request to "application/octet-stream."

  The binary formatter offers superior performance in comparison to the SOAP formatter.

- **SOAP Formatter**. This uses the **SoapFormatter** class to package method calls into a SOAP message. The content type is set to "text/xml" in the HTTP request and is posted to the server with an HTTP POST.

Anatomy of a Request When Hosting in ASP.NET

Remote object endpoints are addressed by URLs that end with the .rem or .soap file name extension, for example http://someserver/vDir/remoteobject.soap. When a request for a remote object (with the extension .rem or .soap), is received by IIS, it is mapped (within IIS) to the ASP.NET ISAPI extension (Aspnet_isapi.dll). The ISAPI extension forwards the request to an application domain within the ASP.NET worker process (Aspnet_wp.exe). The sequence of events is shown in Figure 11.2.

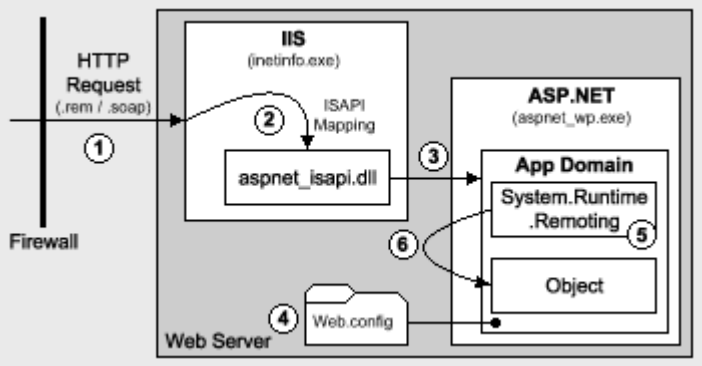

Figure 11.2. Server-side processing

Figure 11.2 shows the following sequence of events:

1. A .soap or .rem request is received over HTTP and is mapped to a specific virtual directory on the Web server.

2. IIS checks the .soap/.rem mapping and maps the file extension to the ASP.NET ISAPI extension, Aspnet_isapi.dll.

3. The ISAPI extension transfers the request to an application domain inside the ASP.NET worker process (Aspnet_wp.exe). If this is the first request directed at this application, a new application domain is created.

4. The **HttpRemotingHandlerFactory** handler is invoked and the remoting infrastructure reads the <**system.runtime.remoting**> section in the Web.config that controls the server-side object configuration (for example, single-call or singleton parameters) and authorization parameters (from the <**authorization**> element).

5. The remoting infrastructure locates the assembly that contains the remote object and instantiates it.

6. The remoting infrastructure reads the HTTP headers and the data stream, and then invokes the method on the remote object.

   Note During this process, ASP.NET calls the normal sequence of event handlers. You can optionally implement one or more of these in Global.asax. For example, **BeginRequest**, **AuthenticationRequest**, **AuthorizeRequest**, and so on. By the time the request reaches the remote object method, the **IPrincipal** object that represents the authenticated user is stored in **HttpContext.User** (and **Thread.CurrentPrincipal**) and is available for authorization. For example, by using principal permission demands and programmatic role checks.

ASP.NET and the HTTP Channel

Remoting does not have its own security model. Authentication and authorization between the client (proxy) and server (remote object) is performed by the channel and host process. You can use the following combination of hosts and channels:

- **A custom executable and the TCP Channel**. This combination does not provide any inbuilt security features.
- **ASP.NET and the HTTP Channel**. This combination provides authentication and authorization through the underlying ASP.NET and IIS security features.

Objects hosted within ASP.NET benefit from the underlying security features of ASP.NET and IIS. These include:

- **Authentication Features**. Windows authentication is configured within Web.config:

```
<authentication mode="Windows"/>
```

The settings in IIS control what type of HTTP authentication is used.

Common HTTP headers are used to authenticate requests. You can supply credentials for the client by configuring the remote object proxy or you can use default credentials.

You cannot use Forms or Passport authentication because the channel does not provide a way to allow the client to access cookies, which is a requirement for both of these authentication mechanisms. Also, Forms and Passport require a redirect to a logon page that requires client interaction. Remote, server side objects are designed for non-interactive use.

- **Authorization Features**. Clients are authorized using standard ASP.NET authorization techniques.

  Configurable authorization options include:

  - URL authorization.
  - File authorization (this requires specific configuration, as described in "Using File Authorization" later in this chapter).

  Programmatic authorization options include:

  - Principal permission demands (declarative and imperative).
  - Explicit role checks using **IPrincipal.IsInRole**.

- **Secure Communication Features**. SSL (and/or IPSec) should be used to secure the transport of data between the client and server.

More information

- For more information about the authentication and authorization features provided by ASP.NET and IIS, see Chapter 8, ASP.NET Security.
- For information about how to host an object in ASP.NET/IIS, see article Q312107, HOW TO: Host a Remote Object in Microsoft Internet Information Services, in the Microsoft Knowledge Base.

.NET Remoting Gatekeepers

The authorization points (or gatekeepers) available to a remote object hosted by ASP.NET are:

- **IIS**. With anonymous authentication turned off, IIS only permits requests from users that it can authenticate either in its domain or in a trusted domain. IIS also provides IP address and DNS filtering.
- **ASP.NET**

- **UrlAuthorizationModule**. You can configure <**authorization**> elements within your application's Web.config to control which users and groups of users should have access to the application. Authorization is based on the **IPrincipal** object stored in **HttpContext.User**.
- **FileAuthorizationModule**. The **FileAuthorizationModule** is available to remote components, although this requires specific configuration, as described in "Using File Authorization" later in this chapter.

  Note Impersonation is not required for File authorization to work.

  The **FileAuthorizationModule** class only performs access checks against the requested file or URI (for example .rem and .soap), and not for files accessed by code within the remote object.

- **Principal Permission Demands and Explicit Role Checks**. In addition to the IIS and ASP.NET configurable gatekeepers, you can also use principal permission demands (declaratively or imperatively) as an additional fine-grained access control mechanism. Principal permission checks allow you to control access to classes, methods, or individual code blocks based on the identity and group membership of individual users, as defined by the **IPrincipal** object attached to the current thread.

  Note Principal permission checks used to demand role membership are different from calling **IPrincipal.IsInRole** to test role membership. The former results in an exception if the caller is not a member of the specified role, while the latter simply returns a Boolean value to confirm role membership.

  With Windows authentication, ASP.NET automatically attaches a **WindowsPrincipal** object that represents the authenticated user to the current Web request (using **HttpContext.User**).

Authentication

When you use remoting in conjunction with an ASP.NET Web application client, authentication occurs within the Web application and at the remote object host. The available authentication options for the remote object host depend on the type of host.

Hosting in ASP.NET

When objects are hosted in ASP.NET the HTTP channel is used to communicate method calls between the client-side proxy and the server. The HTTP channel uses the HTTP protocol to authenticate the remote object proxy to the server.

The following list shows the range of authentication options available when you host inside ASP.NET:

- **IIS Authentication Options**. Anonymous, Basic, Digest, Windows Integrated and Certificate.
- **ASP.NET Authentication Options**. Windows authentication or None (for custom authentication implementations).

  Note Forms and Passport authentication cannot be used directly by .NET Remoting. Calls to remote objects are designed to be non-interactive. If the client of the remote object is a .NET Web application, the Web application can use Forms and Passport authentication and pass credentials explicitly to the remote object. This type of scenario is discussed further in the "Flowing the Original Caller" section later in this chapter.

Hosting in a Windows Service

When objects are hosted in a Windows service, the TCP channel is used to communicate method calls between the client and server. This uses raw socket-based communications. Because there is no authentication provided with sockets, there is no way for the server to authenticate the client.

In this scenario, the remote object must use custom authentication.

Custom authentication

For simple custom authentication, the remote object can expose a **Login** method which accepts a user name and password. The credentials can be validated against a store, a list of roles retrieved, and a token sent back to the

client to use on subsequent requests. When the token is retrieved at the server it is used to create an **IPrincipal** object (with roles) which is stored in **Thread.CurrentPrincipal**, where it is used for authorization purposes.

Other examples of custom authentication include creating a custom transport channel sink that uses an inter-process communication channel that provides authentication, such as named pipes, or creating a channel sink that performs authentication using the Windows Security Service Provider Interface (SSPI).

More information

- For information about how to host an object in a Windows service, see How To: Host a Remote Object in a Windows Service in the Reference section of this guide.
- For more information about sinks and sink chains, search for see the section of the .NET Framework on Sinks and Sink Chains in the MSDN Library.
- For more information about how to create a custom authentication solution that uses SSPI, see the MSDN article .NET Remoting Security Solution, Part 1: Microsoft.Samples.Security.SSPI Assembly.

Note The implementation in this article is a sample and not a product tested and supported by Microsoft.

Authorization

When objects are hosted by ASP.NET and the HTTP channel is used for communication, the client can be authenticated and authorization can be controlled by the following mechanisms:

- URL authorization
- File authorization
- Principal permission demands (declarative and imperative)
- **IPrincipal.IsInRole** checks in code

When objects are hosted in a Windows service, there is no authentication provided by the TCP channel. As a result, you must perform custom authentication and then perform authorization by creating an **IPrincipal** object and storing it in **Thread.CurrentPrincipal**.

You can then annotate your remote object's methods with declarative principal permission demand checks, like the one shown below.

```
[PrincipalPermission(SecurityAction.Demand,

                     Role="Manager")]

void SomeMethod()

{

}
```

Within your object's method code, imperative principal permission demands and explicit role checks using **IPrincipal.IsInRole** can also be used.

Using File Authorization

You may want to use built-in Windows access control to secure the remote object as a securable Windows resource. Without File authorization (using Windows ACLs), you only have URL authorization.

To use the **FileAuthorizationModule** to authorize access to remote object endpoints (identified with .rem or .soap URLs), you must create a physical file with the .rem or .soap extension within your application's virtual directory.

Note The .rem and .soap extensions are used by IIS to map requests for object endpoints to the ASP.NET ISAPI extension (aspnet_isapi.dll). They do not usually exist as physical files.

To configure File authorization for .NET Remoting

1. Create a file with the same name as the objectUri (for example, RemoteMath.rem) in the root of the application's virtual directory.
2. Add the following line to the top of the file and save the file.

```
<%@ webservice class="YourNamespace.YourClass" ... %>
```

4. Add an appropriately configured ACL to the file using Windows Explorer.

Note You can obtain the objectUri from the web.config file used to configure the remote object on the server. Look for the <**wellknown**> element, as shown in the following example.

```
<wellknown mode="SingleCall" objectUri="RemoteMath.rem"
type="RemotingObjects.RemoteMath, RemotingObjects, Version=1.0.000.000
Culture=neutral, PublicKeyToken=4b5ae668c251b606"/>
```

More information

- For more information about these authorization mechanisms, see Chapter 8, ASP.NET Security.
- For more information about principal permission demands, see Chapter 8, ASP.NET Security.

Authentication and Authorization Strategies

In many applications that use .NET Remoting, the remote objects are used to provide business functionality within the application's middle tier and this functionality is called by ASP.NET Web applications. This arrangement is shown in Figure 11.3.

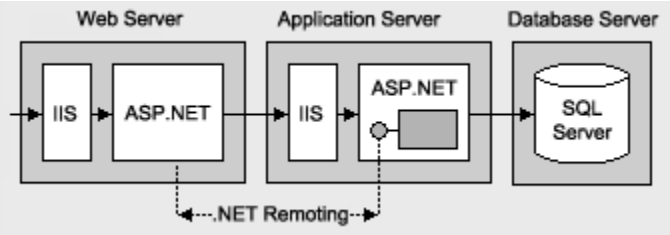

Figure 11.3. Remote objects called by an ASP.NET Web application

In this scenario, the IIS and ASP.NET gatekeepers available to the Web application can be used to secure access to the client-side proxy, and the IIS and ASP.NET gatekeepers available to the ASP.NET host on the remote application server are available to secure access to the remote object.

There are essentially two authentication and authorization strategies for remote objects that are accessed by .NET Web applications.

- You can authenticate and authorize callers at the Web server and then flow the caller's security context to the remote object by using impersonation. This is the impersonation/delegation model.

  With this approach you use an IIS authentication mechanism that allows you to delegate the caller's security context, such as Kerberos, Basic, or Forms authentication (the latter two allow the Web application to access the caller's credentials) and explicitly flow credentials to the remote object using the remote object's proxy.

  The ASP.NET configurable and programmatic gatekeepers (including URL authorization, File authorization, principal permission demands, and .NET roles) are available to authorize individual callers within the remote object.

- You can authenticate and authorize callers at the Web server and then use a trusted identity to communicate with the remote object. This is the trusted subsystem model.

  This model relies on the Web application to authenticate and properly authorize callers before invoking the remote object. Any requests received by the remote object from the trusted identity projected from the Web application are allowed to proceed.

More Information

- For more information about the impersonation/delegation and trusted subsystem models, see Choosing an Authentication Mechanism in Chapter 3, "Authentication and Authorization."
- For more information about using the original caller model with remoting, Flowing the Original Caller later in this chapter.
- For more information about using the trusted subsystem model with remoting, see Trusted Subsystem later in this chapter.

Accessing System Resources

For details about accessing system resources (for example, the event log and the registry) from a remote object hosted by ASP.NET, see Accessing System Resources in Chapter 8, "ASP.NET Security." The approaches and restrictions discussed in Chapter 8 also apply to remote objects hosted by ASP.NET.

Accessing Network Resources

When you access network resources from a remote object, you need to consider the identity that is used to respond to network authentication challenges from the remote computer. You have three options:

- **The Process Identity (this is the default)**. If you host within ASP.NET, the identity used to run the ASP.NET worker process and defined by the <**processModel**> element in Machine.config determines the security context used for resource access.

  If you host within a Windows service, the identity used to run the service process (configured with the Services MMC snap-in) determines the security context used for resource access.

- **A Fixed Service Identity**. For example, one created by calling **LogonUser**.

  Note Don't confuse this service identity with the identity used to run a Windows service. A fixed service identity refers to a Windows user account created specifically for the purposes of accessing resources from an application.

- **The Original Caller Identity**. With ASP.NET configured for impersonation, or programmatic impersonation used within a Windows service.

For details about the relative merits of each of these approaches, together with configuration details, see Accessing Network Resources in Chapter 8, "ASP.NET Security."

Passing Credentials for Authentication to Remote Objects

When a client process calls a remote object, it does so by using a proxy. This is a local object that exposes the same set of methods as the target object.

Specifying Client Credentials

If the remote object is hosted within ASP.NET and is configured for Windows authentication, you must specify the credentials to be used for authentication using the credentials property of the channel. If you do not explicitly set credentials, the remote object is called without any credentials. If Windows authentication is required, this will result in an HTTP status 401, which is an access denied response.

Using DefaultCredentials

If you want to use the credentials of the process that hosts the remote object proxy (or the current thread token, if the thread that calls the proxy is impersonating), you should set the credentials property of the channel to the **DefaultCredentials** maintained by the process credential cache.

You can either specify the use of **DefaultCredentials** in a configuration file or set the credentials programmatically.

Explicit configuration

Within the client application configuration file (Web.config, if the client application is an ASP.NET Web application) set the **useDefaultCredentials** attribute on the <**channel**> element to **true** in order to specify that the proxy should use **DefaultCredentials** when it communicates with the remote object.

```
<channel ref="http" useDefaultCredentials="true" />
```

Programmatic configuration

For programmatic configuration, use the following code to establish the use of **DefaultCredentials** programmatically.

```
IDictionary channelProperties;

channelProperties = ChannelServices.GetChannelSinkProperties(proxy);

channelProperties ["credentials"] = CredentialCache.DefaultCredentials;
```

Using specific credentials

To use a specific set of credentials for authentication when you call a remote object, disable the use of default credentials within the configuration file, by using the following setting.

```
<channel ref="http" useDefaultCredentials="false" />
```

Note Programmatic settings always override the settings in the configuration file.

Then, use the following code to configure the proxy to use specific credentials.

```
IDictionary channelProperties =

                        ChannelServices.GetChannelSinkProperties(proxy);

NetworkCredential credentials;

credentials = new NetworkCredential("username", "password", "domain");

ObjRef objectReference = RemotingServices.Marshal(proxy);

Uri objectUri = new Uri(objectReference.URI);

CredentialCache credCache = new CredentialCache();

// Substitute "authenticationType" with "Negotiate", "Basic",

// "Digest",

// "Kerberos" or "NTLM"

credCache.Add(objectUri, "authenticationType", credentials);
```

```
channelProperties["credentials"] = credCache;

channelProperties["preauthenticate"] = true;
```

Always request a specific authentication type

You should always request a specific authentication type by using the **CredentialCache.Add** method, as illustrated above. Avoid direct use of the **NetworkCredential** class as shown in the following code.

```
IDictionary providerData =
ChannelServices.GetChannelSinkProperties(yourProxy);

providerData["credentials"] = new NetworkCredential(uid, pwd);
```

This should be avoided in production code because you have no control over the authentication mechanism used by the remote object host and as a result you have no control over how the credentials are used.

For example, you may expect a Kerberos or NTLM authentication challenge from the server but instead you may receive a Basic challenge. In this case, the supplied user name and password will be sent to the server in clear text form.

Set the preauthenticate property

The proxy's **preauthenticate** property can be set to true or false. Set it to true (as shown in the above code) to supply specific authentication credentials to cause a **WWW-Authenticate** HTTP header to be passed with the initial request. This stops the Web server denying access on the initial request, and performing authentication on the subsequent request.

Using the connectiongroupname property

If you have an ASP.NET Web application that connects to a remote component (hosted by ASP.NET) and flows the security context of the original caller (by using **DefaultCredentials** and impersonation or by setting explicit credentials, as shown above), you should set the **connectiongroupname** property of the channel within the Web application. This is to prevent a new, unauthenticated client from reusing an old, authenticated connection to the remote component that is associated with a previous client's authentication credentials. Connection reuse can occur as a result of HTTP KeepAlives and authentication persistence which is enabled for performance reasons within IIS.

Set the **connectiongroupname** property to an identifier (such as the caller's user name) that distinguishes one caller from the next.

```
channelProperties["connectiongroupname"] = userName;
```

Note You do not need to set the **connectiongroupname** property if the original caller's security context does not flow through the Web application and onto the remote component, but connects to the remote component using a fixed identity (such as the Web application's ASP.NET process identity). In this scenario, the connection security context remains constant from one caller to the next.
The next version of the .NET Framework will support connection pooling based on the SID of the thread that calls the proxy object, which will help to address the problem described above, if the Web application is impersonating the caller. Pooling will be supported for .NET Remoting clients and not for Web services clients.

Flowing the Original Caller

This section describes how you can flow the original caller's security context through an ASP.NET Web application and onto a remote component hosted by ASP.NET on a remote application server. You may need to do this in order to support per-user authorization within the remote object or within subsequent downstream subsystems (for example databases).

In Figure 11.4, the security context of the original caller (Bob) flows through the front-end Web server that hosts an ASP.NET Web application, onto the remote object, hosted by ASP.NET on a remote application server, and finally through to a back-end database server.

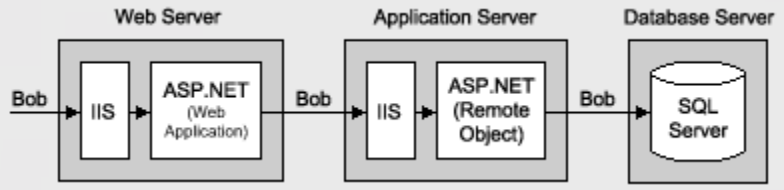

Figure 11.4. Flowing the original caller's security context

In order to flow credentials to a remote object, the remote object client (the ASP.NET Web application in this scenario) must configure the object's proxy and explicitly set the proxy's **credentials** property, as described in "Passing Credentials for Authentication to Remote Objects" earlier in this chapter.

Note **IPrincipal** objects do not flow across .NET Remoting boundaries.

There are two ways to flow the caller's context:

- **Pass default credentials and use Kerberos authentication (and delegation)**. This approach requires that you impersonate within the ASP.NET Web application and configure the remote object proxy with **DefaultCredentials** obtained from the impersonated caller's security context.
- **Pass explicit credentials and use Basic or Forms authentication**. This approach does not require impersonation within the ASP.NET Web application. Instead, you programmatically configure the remote object proxy with explicit credentials obtained from either, server variables (with Basic authentication), or HTML form fields (with Forms authentication) that are available to the Web application. With Basic or Forms authentication, the username and password are available to the server in clear text.

Default Credentials with Kerberos Delegation

To use Kerberos delegation, all computers (servers and clients) must be running Windows 2000 or later. Additionally, client accounts that are to be delegated must be stored in Active Directory™ directory service and must not be marked as "Sensitive and cannot be delegated."

The following tables show the configuration steps required on the Web server and application server.

Configuring the Web server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web application's virtual root directory Enable Windows Integrated Authentication for the Web application's virtual root	 Kerberos authentication will be negotiated assuming clients and server are running Windows 2000 or above. **Note:** If you are using Microsoft Internet Explorer 6 on Windows 2000, it defaults to NTLM authentication instead of the required Kerberos authentication. To enable Kerberos delegation, see article Q299838, Unable to Negotiate Kerberos Authentication after upgrading to Internet Explorer 6, in the Microsoft Knowledge Base.
Configure ASP.NET	
Step	**More Information**
Configure your ASP.NET Web application to use Windows authentication	Edit Web.config in your Web application's virtual directory. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure your ASP.NET Web	Edit Web.config in your Web application's virtual directory.

application for impersonation	Set the <**identity**> element to: `<identity impersonate="true" />`
Configure Remoting (Client Side Proxy)	
Step	**More Information**

Configure the remote object proxy to use default credentials for all calls to the remote object	Add the following entry to Web.config:
	
	`<channel ref="http"`
	
	`useDefaultCredentials="true" />`
	
	Credentials will be obtained from the Web application's thread impersonation token.

Configuring the remote application server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your Web application's virtual root directory Enable Windows Integrated Authentication for the Web application's virtual root	
Configure ASP.NET (Remote Object Host)	
Step	**More Information**
Configure ASP.NET to use Windows authentication	Edit Web.config in the application's virtual directory. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure ASP.NET for impersonation	Edit Web.config in the application's virtual directory. Set the <**identity**> element to: `<identity impersonate="true" />` **Note:** This step is only required if you want to flow the original caller's security context through the remote object and onto the next, downstream subsystem (for example, database). With impersonation enabled here, resource access (local and remote) uses the impersonated original caller's security context. If your requirement is simply to allow per-user authorization checks in the remote object, you do not need to impersonate here.

More information

For more information about Kerberos delegation, see How To: Implement Kerberos Delegation for Windows 2000 in the Reference section of this guide.

Explicit Credentials with Basic or Forms Authentication

As an alternative to Kerberos delegation, you can use Basic or Forms authentication at the Web application to capture the client's credentials and then use Basic (or Integrated Windows) authentication to the remote object.

With this approach, the client's clear text credentials are available to the Web application. These can be passed to the remote object through the remote object proxy. For this, you must include code in the Web application to retrieve the client's credentials and configure the remote object proxy.

Basic authentication

With Basic authentication, the original caller's credentials are available to the Web application in server variables. The following code shows how to retrieve them and configure the remote object proxy.

```
// Retrieve client's credentials (available with Basic

// authentication)

string pwd = Request.ServerVariables["AUTH_PASSWORD"];

string uid = Request.ServerVariables["AUTH_USER"];

// Associate the credentials with the remote object proxy

IDictionary channelProperties =

                         ChannelServices.GetChannelSinkProperties(proxy);

NetworkCredential credentials;

credentials = new NetworkCredential(uid, pwd);

ObjRef objectReference = RemotingServices.Marshal(proxy);

Uri objectUri = new Uri(objectReference.URI);

CredentialCache credCache = new CredentialCache();

credCache.Add(objectUri, "Basic", credentials);

channelProperties["credentials"] = credCache;

channelProperties["preauthenticate"] = true;
```

Note The **NetworkCredential** constructor shown in the above code is supplied with the user ID and password. To avoid hard coding the domain name, a default domain can be configured at the Web server within IIS when you configure Basic authentication.

Forms authentication

With Forms authentication, the original caller's credentials are available to the Web application in form fields (rather than server variables). In this case, use the following code.

```
// Retrieve client's credentials from the logon form

string pwd = txtPassword.Text;

string uid = txtUid.Text;
```

```
// Associate the credentials with the remote object proxy

IDictionary channelProperties =

                          ChannelServices.GetChannelSinkProperties(proxy);

NetworkCredential credentials;

credentials = new NetworkCredential(uid, pwd);

ObjRef objectReference = RemotingServices.Marshal(proxy);

Uri objectUri = new Uri(objectReference.URI);

CredentialCache credCache = new CredentialCache();

credCache.Add(objectUri, "Basic", credentials);

channelProperties["credentials"] = credCache;

channelProperties["preauthenticate"] = true;
```

The following tables show the configuration steps required on the Web server and application server.

Configuring the Web server

Configure IIS	
Step	**More Information**
To use Basic authentication, disable Anonymous access for your Web application's virtual root directory and select Basic authentication - or - To use Forms authentication, enable anonymous access	Both Basic and Forms authentication should be used in conjunction with SSL to protect the clear text credentials sent over the network. If you use Basic authentication, SSL should be used for all pages (not just the initial logon page) because Basic credentials are transmitted with every request. Similarly, SSL should be used for all pages if you use Forms authentication to protect the clear text credentials on the initial logon and to protect the authentication ticket passed on subsequent requests.
Configure ASP.NET	
Step	**More Information**

If you use Basic authentication, configure your ASP.NET Web application to use Windows authentication
 	
- or -
 	
If you use Forms authentication, configure your ASP.NET Web application to use Forms authentication
	Edit Web.config in your Web application's virtual directory.
	Set the <**authentication**> element to:
	
	`<authentication mode="Windows" />`
	
	- or -
	
	Edit Web.config in your Web application's virtual directory.
	Set the <**authentication**> element to:
	
	`<authentication mode="Forms" />`

Disable impersonation within the ASP.NET Web application	Edit Web.config in your Web application's virtual directory. Set the <**identity**> element to:

	`<identity impersonate="false" />` **Note:** This is equivalent to having no <**identity**> element. Impersonation is not required because the user's credentials will be passed explicitly to the remote object through the remote object proxy.
Configure Remoting (Client Side Proxy)	
Step	**More Information**

Configure the remoting proxy to not use default credentials for all calls to the remote object	Add the following entry to Web.config:
	
	`<channel ref="http"`
	`useDefaultCredentials="false" />`
	
	You do not want default credentials to be used (because the Web application is configured not to impersonate; this would result in the security context of the ASP.NET process identity being used).

Write code to capture and explicitly set the credentials on the remote object proxy	Refer to the code fragments shown earlier.

Configuring the application server

Configure IIS	
Step	**More Information**
Disable Anonymous access for your application's virtual root directory Enable Basic authentication	**Note:** Basic authentication at the application server (remote object), allows the remote object to flow the original caller's security context to the database (because the caller's user name and password are available in clear text and can be used to respond to network authentication challenges from the database server). If you don't need to flow the original caller's security context beyond the remote object, consider configuring IIS at the application server to use Windows Integrated authentication because this provides tighter security—credentials are not passed across the network and are not available to the application.
Configure ASP.NET (Remote Object Host)	
Step	**More Information**
Configure ASP.NET to use Windows authentication	Edit Web.config in the application's virtual directory. Set the <**authentication**> element to: `<authentication mode="Windows" />`
Configure ASP.NET for impersonation	Edit Web.config in the application's virtual directory. Set the <**identity**> element to:

	`<identity impersonate="true" />` **Note:** This step is only required if you want to flow the original caller's security context through the remote object and onto the next, downstream subsystem (for example, database). With impersonation enabled here, resource access (local and remote) uses the impersonated original caller's security context. If your requirement is simply to allow per-user authorization checks in the remote object, you do not need to impersonate here.

Trusted Subsystem

The trusted subsystem model provides an alternative (and simpler to implement) approach to flowing the original caller's security context. In this model, a trust boundary exists between the remote object host and Web application. The remote object trusts the Web application to properly authenticate and authorize callers, prior to letting requests proceed to the remote object. No authentication of the original caller occurs within the remote object host. The remote object host authenticates the fixed, trusted identity used by the Web application to communicate with the remote object. In most cases, this is the process identity of the ASP.NET Web application.

The trusted subsystem model is shown in Figure 11.5. This diagram also shows two possible configurations. The first uses the ASP.NET host and the HTTP channel, while the second uses a Windows service host and the TCP channel.

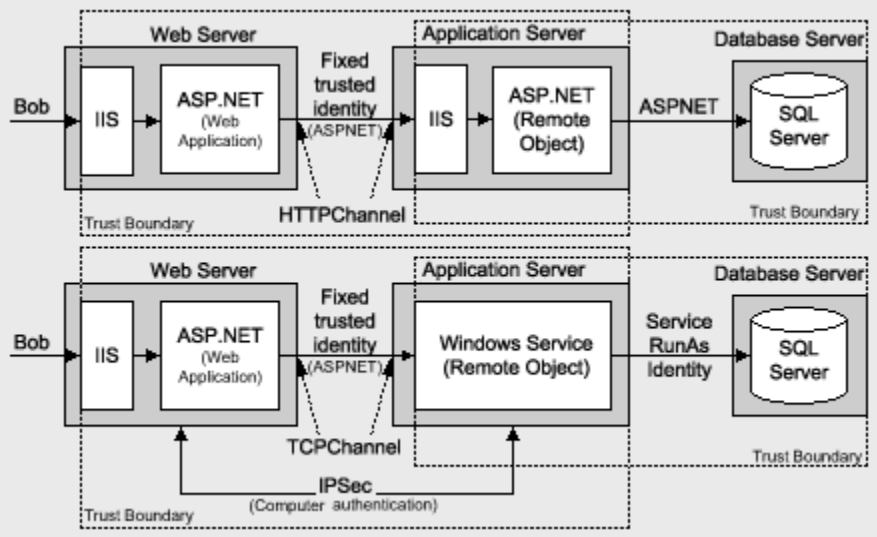

Figure 11.5. The trusted subsystem model

Flowing the Caller's Identity

If you use the trusted subsystem model, you may still need to flow the original caller's identity (name, not security context), for example, for auditing purposes at the database.

You can flow the identity at the application level by using method and stored procedure parameters and trusted query parameters (as shown in the following example) can be used to retrieve user-specific data from the database.

```
SELECT x,y,z FROM SomeTable WHERE UserName = "Bob"
```

Choosing a Host

The trusted subsystem model means that the remote object host does not authenticate the original callers. However, it must still authenticate (and authorize) its immediate client (the ASP.NET Web application in this scenario), to prevent unauthorized applications issuing requests to the remote object.

If you host within ASP.NET and use the HTTP channel, you can use Windows Integrated authentication to authenticate the ASP.NET Web application process identity.

If you host within a Windows service, you can use the TCP channel which offers superior performance but no authentication capabilities. In this scenario, you can use IPSec between the Web server and application server. An IPSec policy can be established that only allows the Web server to communicate with the application server.

Configuration Steps

The following tables show the configuration steps required on the Web server and application server.

Configuring the Web server

Configure IIS	
Step	**More Information**
Configure IIS authentication	The Web application can use any form of authentication to authenticate the original callers.
Configure ASP.NET	
Step	**More Information**

Configure authentication and make sure impersonation is disabled	Edit Web.config in your Web application's virtual directory.
	Set the <**authentication**> element to "Windows", "Forms" or "Passport."
	`<authentication mode="Windows|Forms|Passport" />`
	Set the <**identity**> element to:
	`<identity impersonate="false" />`
	(OR remove the <**identity**> element)

Reset the password of the ASPNET account used to run ASP.NET OR create a least privileged domain account to run ASP.NET and specify account details on the <**processModel**> element within Web.config	For more information about how to access network resources (including remote objects) from ASP.NET and about choosing and configuring a process account for ASP.NET, see Accessing Network Resources and Process Identity for ASP.NET in Chapter 8, "ASP.NET Security."
Configure Remoting (Client Side Proxy	
Step	**More Information**

Configure the remoting proxy to use default credentials for all calls to the remote object	Add the following entry to Web.config:
	`<channel ref="http"`
	`useDefaultCredentials="true" />`
	Because the Web application is not impersonating, using default credentials results in the use of the ASP.NET process identity for all calls to the remote

	object.

Configuring the application server

The following steps apply if you are using an ASP.NET host.

Configure IIS	
Step	**More Information**
Disable Anonymous access for your application's virtual root directory Enable Windows Integrated authentication	
Configure ASP.NET (Remote Object Host)	
Step	**More Information**
Configure ASP.NET to use Windows authentication	Edit Web.config in the application's virtual directory. et the <**authentication**> element to: `<authentication mode="Windows" />`
Disable impersonation	Edit Web.config in the application's virtual directory. Set the <**identity**> element to: `<identity impersonate="false" />`

Using a Windows service host

If you are using a Windows service host process, you must create a Windows account to run the service. This security context provided by this account will be used by the remote object for all local and remote resource access.

To access a remote Microsoft SQL Server™ database (using Windows authentication), you can use a least privileged domain account, or use a least privileged local account and then create a duplicated account (with the same user name and password) on the database server.

Secure Communication

Secure communication is related to guaranteeing the integrity and confidentiality of messages as they flow across the network. You can use a platform-based approach to secure communication and use SSL or IPSec, or you can use a message-level approach and develop a custom encryption sink to encrypt the entire message, or selected parts of a message.

Platform Level Options

The two platform-level options to consider for securing the data passed between a client and remote component are:

- SSL
- IPSec

If you host remote objects in ASP.NET, you can use SSL to secure the communication channel between client and server. This requires a server authentication certificate on the computer that hosts the remote object.

If you host remote objects in a Windows service, you can use IPSec between the client and host (server) computers, or develop a custom encryption sink.

Message-Level options

Due to the extensible nature of the .NET Remoting architecture, you can develop your own custom sinks and plug them into the processing pipeline. To provide secure communication, you can develop a custom sink that encrypts and decrypts the message data sent to and from the remote object.

The advantage of this approach is that it allows you to selectively encrypt parts of a message. This is in contrast to the platform-level approaches that encrypt all the data sent between client and server.

More information

For more information about SSL and IPSec, see Chapter 4, Secure Communication.

Choosing a Host Process

Objects that are to be accessed remotely must run in a host executable. The host listens for incoming requests and dispatches calls to objects. The type of host selected influences the message transport mechanism called a channel. The type of channel that you select influences the authentication, authorization, secure communication, and performance characteristics of your solution.

The HTTP channel provides better security options, but the TCP channel provides superior performance.

You have the following main options for hosting remote objects:

- Host in ASP.NET
- Host in a Windows Service
- Host in a Console Application

Recommendation

To take advantage of the security infrastructure provided by ASP.NET and IIS, it is recommended from a security standpoint to host remote objects in ASP.NET. This requires clients to communicate with the remote objects over the HTTP channel. ASP.NET and IIS authentication, authorization, and secure communication features are available to remote objects that are hosted in ASP.NET.

If performance (and not security) is the primary concern, consider hosting remote objects in Windows services.

Hosting in ASP.NET

When you host a remote object in ASP.NET:

- The object is accessed using the HTTP protocol.
- It has an endpoint that is accessible by a URL.
- It exists in an application domain inside the Aspnet_wp.exe worker process.
- It inherits the security features offered by IIS and ASP.NET.

Advantages

If you host remote objects in IIS, you benefit from the following advantages:

- Authentication, authorization, and secure communication features provided by IIS and ASP.NET are immediately available.
- You can use the auditing features of IIS.
- The ASP.NET worker process is always running.

- You have a high degree of control over the hosting executable through the <**processModel**> element in Machine.config. You can control thread management, fault tolerance, memory management, and so on.
- You can create a Web services façade layer in front of the remote object.

Disadvantages

If you use ASP.NET to host remote objects, you should be aware of the following disadvantages:

- It requires the use of the HTTP channel which is slower than the TCP channel.
- User profiles are not loaded by ASP.NET. Various encryption techniques (including DPAPI) may require user profiles.
- If the object is being accessed from code running in an ASP.NET Web application, you may have to use Basic authentication.

Hosting in a Windows Service

When you host a remote object in a Windows service, the remote object lives in an application domain contained within the service process. You cannot use the HTTP channel and must use the TCP channel. The TCP channel supports the following security features:

- **Authentication Features**

  You must provide a custom authentication solution. Options include:

  - **Using the underlying authentication services of the SSPI**. You can create a channel sink that uses the Windows SSPI credential and context management APIs to authenticate the caller and optionally impersonate the caller. The channel-sink sits on top of the TCP channel. The SSPI in conjunction with the TCP channel allows the client and server to exchange authentication information. After authentication the client and server can send messages ensuring confidentiality and integrity.
  - **Using an underlying transport that supports authentication, for example, named pipes**. The named pipe channel uses named pipes as the transport mechanism. This provides authentication of the caller and also introduces Windows ACL-based security on the pipe and also impersonation of the caller.

- **Authorization Features**

  Authorization is possible only if you implement a custom authentication solution.

  - If you are able to impersonate the user (for example, by using an underlying named pipe transport), you can use **WindowsPrincipal.IsInRole**.
  - If you are able to create an **IPrincipal** object to represent the authenticated client, you can use .NET roles (through principal permission demands and explicit role checking using **IPrincipal.IsInRole**)

- **Secure Communication Features**

  You have two options:

  - Use IPSec to secure the transport of data between the client and server.
  - Create a custom channel sink that performs asymmetric encryption. This option is discussed later in this chapter.

Advantages

If you host remote objects in Windows services, you benefit from the following advantages:

- High degree of activation control over the host process
- Inherits the benefits of Windows service architecture

- No need to introduce IIS on your application's middle tier
- User profiles are automatically loaded
- Performance is good, as clients communicate over the TCP channel using binary encoded data

Disadvantages

If you use a Windows service to host remote objects, you should be aware of the following disadvantages:

- You must provide custom authentication and authorization solutions.
- You must provide secure communication solutions.
- You must provide auditing solutions.

Hosting in a Console Application

When you host a remote object in a console application, the remote object lives in an application domain contained within the console application process. You cannot use the HTTP channel and must use the TCP channel.

This approach is not recommended for production solutions.

Advantages

There are very few advantages to this approach, although it does mean that IIS is not required on the middle tier. However, this approach is only recommended for development and testing and not for production environments.

Disadvantages

If you host remote objects in a custom executable, you should be aware of the following disadvantages:

- The host must be manually started and runs under the interactive logon session (which is not recommended).
- There is no fault tolerance.
- You must provide custom authentication and authorization.
- There is no auditing capability.

Remoting vs. Web Services

.NET offers many different techniques to allow clients to communicate with remote objects including the use of Web services.

If you need interoperability between heterogeneous systems, a Web services approach that uses open standards such as SOAP, XML, and HTTP is the right choice. On the other hand, if you are creating server to server intranet-based solutions, remoting offers the following features:

- Rich object fidelity because any .NET type (including custom types created using Microsoft C#® development tool and Microsoft Visual Basic® .NET development system) can be remoted.

  This includes classes, class hierarchies, interfaces, fields, properties, methods and delegates, datasets, hash tables, and so on.

- Objects may be marshaled by value and by reference.
- Object lifetime management is lease-based.
- High performance, particularly with the TCP channel and binary formatter.
- It allows you to construct load balanced middle tiers, using network load balancing.

Table 11.2. The major differences between remoting and Web services

Remoting	Web Services
State full or stateless, lease-based object lifetime management	All method calls are stateless
No need for IIS (Although hosting in IIS/ASP.NET is recommended for security)	Must have IIS installed on the server
All managed types are supported	Limited data types are supported. For more information about the types supported by ASP.NET Web services, see the .NET Framework Developer's Guide on MSDN.
Objects can be passed by reference or by value	Objects cannot be passed
Contains an extensible architecture not limited to HTTP or TCP transports	Limited to XML over HTTP
Can plug custom processing sinks into the message processing pipeline	No ability to modify messages
SOAP implementation is limited and can only use RPC encoding	SOAP implementation can use RPC or document encoding and can fully interoperate with other Web service platforms. For more information, see the "Message Formatting and Encoding" section of the Distributed Application Communication article on MSDN.
Tightly coupled	Loosely coupled

Summary

.NET Remoting does not provide its own security model. However, by hosting remote objects in ASP.NET and by using the HTTP channel for communication, remote objects can benefit from the underlying security services provided by IIS and ASP.NET. In comparison, the TCP channel and a custom host executable offers improved performance, but this combination provides no built-in security.

- If you want to authenticate the client, use the HTTP channel, host in ASP.NET, and disable Anonymous access in IIS.
- Use the TCP channel for better performance and if you don't care about authenticating the client.
- Use IPSec to secure the communication channel between client and server if you use the TCP channel. Use SSL to secure the HTTP channel.
- If you need to make trusted calls to a remote resource, host the component in Windows service and not a console application.
- **IPrincipal** objects are not passed across .NET Remoting boundaries. You could consider implementing your own **IPrincipal** class that can be serialized. If you do so, be aware that it would be relatively easy for a rogue client to spoof an **IPrincipal** object and send it to your remote object. Also, be careful of **IlogicalThreadAffinitive** if you implement your own **IPrincipal** class for remoting.
- Never expose remote objects to the Internet. Use Web services for this scenario.

  .NET Remoting should be used on the intranet only. Objects should be accessed from Web applications internally. Even if an object is hosted in ASP.NET, don't expose them to Internet clients, as clients would need to be .NET clients.

Data Access Security

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET
Microsoft® SQL Server™

See the Landing Page for the starting point and complete overview of *Building Secure ASP.NET Applications*.

Summary: This chapter presents recommendations and guidance that will help you develop a secure data access strategy. Topics covered include using Windows authentication from ASP.NET to the database, securing connection strings, storing credentials securely in a database, protecting against SQL injection attacks and using database roles. (33 printed pages)

Contents

When you build Web-based applications, it is essential that you use a secure approach to accessing and storing data. This chapter addresses some of the key data access issues. It will help you:

- Choose between Microsoft® Windows® operating system authentication and SQL authentication when connecting to SQL Server™.
- Store connection strings securely.
- Decide whether to flow the original caller's security context through to the database.
- Take advantage of connection pooling.
- Protect against SQL injection attacks.
- Store credentials securely within a database.

The chapter also presents various trade-offs that relate to the use of roles, for example, roles in the database versus role logic applied in the middle tier. Finally, a set of core recommendations for data access are presented.

Introducing Data Access Security

Figure 12.1 shows key security issues associated with data access.

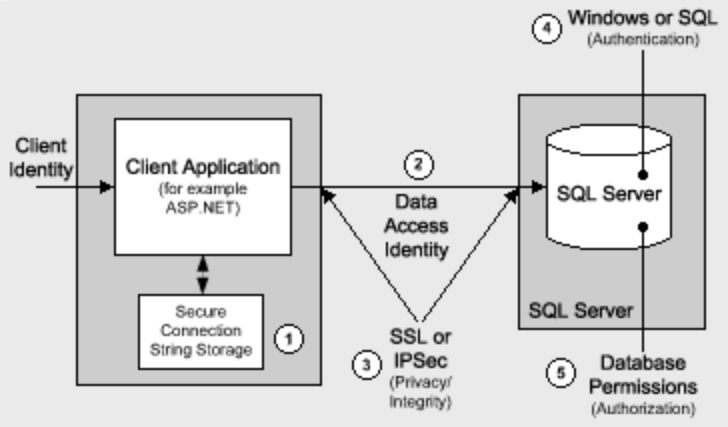

Figure 12.1. Key data access security issues

The key issues shown in Figure 12.1 and discussed throughout the remainder of this chapter are summarized below:

1. **Storing database connection strings securely**. This is particularly significant if your application uses SQL authentication to connect to SQL Server or connects to non-Microsoft databases that require explicit logon credentials. In these cases, connection strings include clear text usernames and passwords.

2. **Using an appropriate identity or identities to access the database**. Data access may be performed by using the process identity of the calling process, one or more service identities, or the original caller's identity (with impersonation/delegation). The choice is determined by your data access model—trusted subsystem or impersonation/delegation.

3. **Securing data that flows across the network**. For example, securing login credentials and sensitive data passed to and from SQL Server.

   Note Login credentials are only exposed on the network if you use SQL authentication, not Windows authentication.

   SQL Server 2000 supports SSL, with server certificates. IPSec can also be used to encrypt traffic between the client computer (for example, a Web or application server) and database server.

4. **Authenticating callers at the database.** SQL Server supports Windows authentication (using NTLM or Kerberos) and SQL authentication (using SQL Server's built-in authentication mechanism).

5. **Authorizing callers at the database.** Permissions are associated with individual database objects. Permissions can be associated with users, groups, or roles.

SQL Server Gatekeepers

Figure 12.2 highlights the key gatekeepers for SQL server data access.

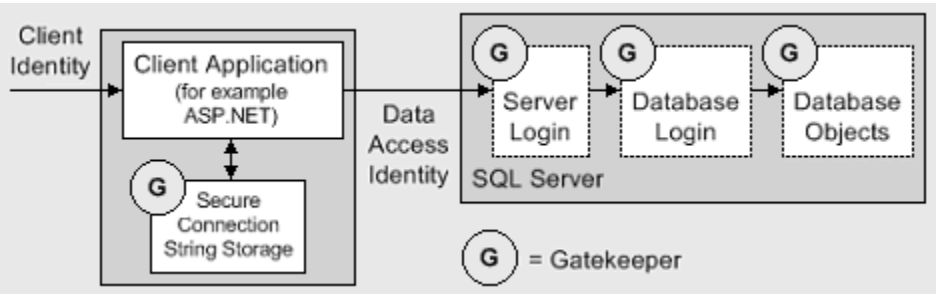

Figure 12.2. SQL Server gatekeepers

The key gatekeepers are:

- The chosen data store used to maintain the database connection string.
- The SQL Server login (as determined by the server name specified in the connection string).
- The database login (as determined by the database name specified in the connection string).
- Permissions attached to individual database objects.

Permissions may be assigned to users, groups, or roles.

Trusted Subsystem vs. Impersonation/Delegation

Granularity of access to the database is a key factor to consider. You must consider whether you need user-level authorization at the database (which requires the impersonation/delegation model), or whether you can use application role logic within the middle tier of your application to authorize users (which implies the trusted subsystem model).

If your database requires user-level authorization, you need to impersonate the original caller. While this impersonation/delegation model is supported, you are encouraged to use the trusted subsystem model, where the original caller is checked at the IIS/ASP.NET gate, mapped to a role, and then authorized based on role membership. System resources for the application are then authorized at the application or role level using service accounts, or using the application's process identity (such as the ASPNET account).

Figure 12.3 shows the two models.

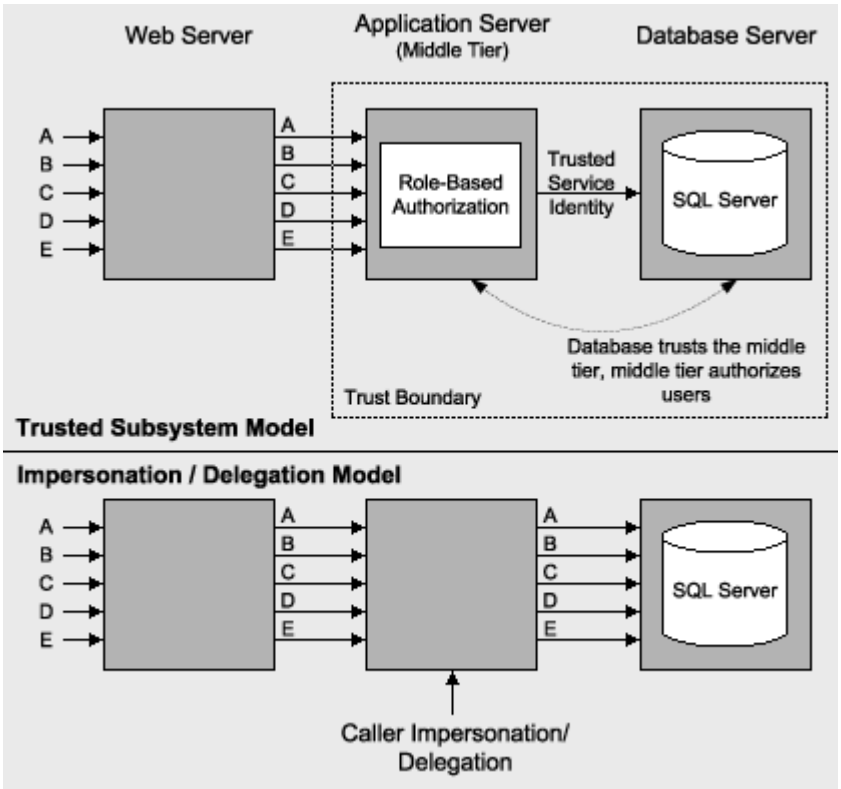

Figure 12.3. The trusted subsystem and impersonation/delegation models for database access

There are a number of key factors that you should consider when connecting to SQL Server for data access. These are summarized below and elaborated upon in subsequent sections:

- **What type of authentication should you use?** Windows authentication offers improved security, but firewalls and non-trusting domain issues may force you to use SQL authentication. If so, you should ensure that your application's use of SQL authentication is as secure as possible, as discussed in the "SQL Authentication" section later in this chapter.

- **Single user role versus multiple user roles**. Does your application need to access SQL using a single account with a fixed set of permissions within the database, or are multiple (role-based) accounts required depending on the user of the application?

- **Caller identity**. Does the database need to receive the identity of the original caller through the call context either to perform authorization or to perform auditing, or can you use one or more trusted connections and pass the original caller identity at the application level?

  For the operating system to flow the original caller's identity, it requires impersonation/delegation in the middle tier. This dramatically reduces the effectiveness of connection pooling. Connection pooling is still enabled, but it results in many small pools (for each separate security context), with little if any reuse of connections.

- **Are you sending sensitive data to and from the database server?** While Windows authentication means that you do not pass user credentials over the network to the database server, if your application's data is sensitive (for example, employee details or payroll data), then this should be secured using IPSec or SSL.

Authentication

This section discusses how you should authenticate clients to SQL Server and how you choose an identity to use for database access within client applications, prior to connecting to SQL Server.

Windows Authentication

Windows authentication is more secure than SQL authentication for the following reasons:

- Credentials are managed for you and the credentials are not transmitted over the network.

- You avoid embedding user names and passwords in connection strings.

- Logon security improves through password expiration periods, minimum lengths, and account lockout after multiple invalid logon requests. This mitigates the threat from dictionary attacks.

Use Windows authentication in the following scenarios:

- You have used the trusted subsystem model and you connect to SQL Server using a single fixed identity. If you are connecting from ASP.NET, this assumes that the Web application is not configured for impersonation.

  In this scenario, use the ASP.NET process identity or a serviced component identity (obtained from the account used to run an Enterprise Services server application).

- You are intentionally delegating the original caller's security context by using delegation (and are prepared to sacrifice application scalability by foregoing database connection pooling).

Consider the following key points when you use Windows authentication to connect to SQL Server:

- Use the principle of least privilege for the ASP.NET process account. Avoid giving the ASP.NET process account the "Act as part of the operating system" privilege to enable **LogonUser** API calls.

- Determine which code requires additional privileges, and place it within serviced components that run in out-of-process Enterprise Services applications.

More information

For more information about accessing network resources from ASP.NET and choosing and configuring an appropriate account to run ASP.NET, see Chapter 8, ASP.NET Security.

Using Windows authentication

You have the following options when you use Windows authentication to connect to SQL Server from an ASP.NET application (or Web service, or remote component hosted by ASP.NET):

- Use the ASP.NET process identity.
- Use fixed identities within ASP.NET.
- Use serviced components.
- Use the **LogonUser** API and impersonating a specific identity.
- Use the original caller's identity.
- Use the anonymous Internet User account.

Recommendation

The recommendation is to configure the local ASP.NET process identity by changing the password to a known value on the Web server and create a mirrored account on the database server by creating a local user with the same name and password. Further details for this and the other approaches are presented below.

Using the ASP.NET process identity

If you connect to SQL Server directly from an ASP.NET application (or Web service, or remote component hosted by ASP.NET), use the ASP.NET process identity. This is a common approach and the application defines the trust boundary, that is, the database trusts the ASP.NET account to access database objects.

You have three options:

- Use mirrored ASPNET local accounts.
- Use mirrored, custom local accounts.
- Use a custom domain account.

Use mirrored ASPNET local accounts

This is the simplest approach and is the one generally used when you own the target database (and can control the administration of local database-server accounts). With this option, you use the ASPNET least-privileged, local account to run ASP.NET and then create a duplicated account on the database server.

Note This approach has the added advantages that it works across non-trusting domains and through firewalls. The firewall may not open sufficient ports to support Windows authentication.

Use mirrored, custom local accounts

This approach is the same as the previous approach except that you don't use the default ASPNET account. This means two things:

- You will need to create a custom local account with appropriate permissions and privileges.

  For more information, see How To: Create a Custom Account to Run ASP.NET in the Reference section of this guide.

- You are no longer using the default account created by the .NET Framework installation process. Your company may have a policy not to use default installation accounts. This can potentially raise the security bar of your application.

  For more information, see the Sans Top 20, Accounts with No Passwords or Weak Passwords.

Use a custom domain account

This approach is similar to the previous one except that you use a least-privileged domain account instead of a local account. This approach assumes that client and server computers are in the same or trusting domains. The main benefit is that credentials are not shared across machines; the machines simply give access to the domain account. Also, administration is easier with domain accounts.

Implementing mirrored ASPNET process identity

In order to use mirrored accounts to connect from ASP.NET to a database, you need to perform the following actions:

- Use User Manager on the Web server to reset the ASPNET account's password to a known strong password value.

  Important If you change the ASPNET password to a known value, the password in the Local Security Authority (LSA) on the local computer will no longer match the account password stored in the Windows Security Account Manager (SAM) database. If you need to revert to the AutoGenerate default, you must do the following:

  Run Aspnet_regiis.exe to reset ASP.NET to its default configuration. For more information, see article Q306005, HOWTO: Repair IIS Mapping After You Remove and Reinstall IIS, in the Microsoft Knowledge Base. When you do this, you get a new account and a new Windows Security Identifier (SID). The permissions for this account are set to their default values. As a result, you need to explicitly reapply permissions and privileges that you had originally set for the old ASPNET account.

- Explicitly set the password in Machine.config.

- ```
  <processModel userName="machine" password="YourStrongPassword"    .
  ```

- You should protect Machine.config from unauthorized access by using Windows ACLs. For example, restrict Machine.config from the IIS anonymous Internet user account.

- Create a mirrored account (with the same name and password) on the database server.

- Within the SQL database, create a server login for the local ASPNET account and then map the login to a user account within the required database. Then create a database user role, add the database user to the role, and configure the appropriate database permissions for the role.

  For more information, see Creating a Least Privilege Database Account later in this chapter.

Connecting to SQL Server using Windows authentication

To connect to SQL Server using Windows authentication

- Within the client application, use a connection string that contains either "Trusted Connection=Yes", or "Integrated Security=SSPI". The two strings are equivalent and both result in Windows authentication (assuming that your SQL Server is configured for Windows authentication). For example:

- ```
  "server=MySQL; Integrated Security=SSPI; database=Northwind"
  ```

  Note The identity of the client making the request (that is, the client authenticated by SQL Server) is determined by the client's thread impersonation token (if the thread is currently impersonating) or the client's current process token.

Using fixed identities within ASP.NET

With this approach, you configure your ASP.NET application to impersonate a specified, fixed identity, by using the following element in Web.config.

```
<identity impersonate="true"

          userName="YourAccount"

          password="YourStrongPassword" />
```

This becomes the default identity that is used when you connect to network resources, including databases.

This approach is not recommended with the .NET Framework version 1.0 for two reasons:

- User names and passwords are in clear text in the Web space (that is, in Web.config in a virtual directory).
- ASP.NET (on Windows 2000) requires the "Act as part of the operating system" privilege. This restriction does not apply for Microsoft Windows Server 2003.

  For more information about this strong privilege, see the Security Briefs column in the August 99 issue of Microsoft Systems Journal.

The .NET Framework version 1.1 will provide an enhancement for this scenario on Windows 2000. Specifically:

- The credentials will be encrypted.
- The logon will be performed by the IIS process, so that ASP.NET does not require the "Act as part of the operating system" privilege.

Using serviced components

You can develop a serviced component specifically to contain data access code. With serviced components, you can access the database by either hosting your component in an Enterprise Services (COM+) server application running under a specific identity, or you can write code that uses the **LogonUser** API to perform impersonation.

Using out of process serviced components raises the security bar because process hops make an attacker's job more difficult, particularly if the processes run with different identities. The other advantage is that you can isolate code that requires more privilege from the rest of the application.

Calling LogonUser and impersonating a specific Windows identity

You should not call **LogonUser** directly from ASP.NET. In Windows 2000, this approach requires you to give the ASP.NET process identity "Act as part of the operating system".

A preferred approach is to call **LogonUser** outside of the ASP.NET process using a serviced component in an Enterprise Services server application, as discussed above.

Using the original caller's identity

For this approach to work, you need to use Kerberos delegation and impersonate the caller to the database, either directly from ASP.NET or from a serviced component.

From ASP.NET add the following to your application's Web.config.

```
<identity impersonate="true" />
```

From a serviced component, call **CoImpersonateClient**.

Using the anonymous Internet user account

As a variation of the previous approach, for scenarios where your application uses Forms or Passport authentication (which implies IIS anonymous authentication), you can enable impersonation within your application's Web.config in order to use the anonymous Internet user account for database access.

```
<identity impersonate="true" />
```

With IIS configured for anonymous authentication, this configuration results in your Web application's code running using the anonymous Internet user's impersonation token. In a Web hosting environment, this has the advantage of allowing you to separately audit and track database access from multiple Web applications.

More Information

- For more information and implementation details about using the original caller's identity, see Flowing the Original Caller to the Database in Chapter 5, "Intranet Security."
- For more information about how to configure IIS to use anonymous user account refer to Chapter 8, ASP.NET Security.

When can't you use Windows authentication?

Certain application scenarios may prevent the use of Windows authentication. For example:

- Your database client and database server are separated by a firewall which prevents Windows authentication.
- Your application needs to connect to one or more databases using multiple identities.
- You are connecting to databases other than SQL Server.
- You don't have a secure way within ASP.NET to run code as a specific Windows user. Either you can't (or won't) forward the original caller's security context, and/or you want to use a dedicated service account rather than grant logons to end users.

  Specifying a user name and password in Machine.config (on the <**processModel**> element) or in Web.config (on the <**identity**> element) in order to run the ASP.NET worker process or your application is less secure than taking explicit steps to protect standard SQL credentials.

In these scenarios, you will have to use SQL authentication (or the database's native authentication mechanism), and you must:

- Protect database user credentials on the application server.
- Protect database user credentials while in transit from the server to the database.

If you do use SQL authentication, there are various ways in which you can make SQL authentication more secure. These are highlighted in the next section.

SQL Authentication

If your application needs to use SQL authentication, you need to consider the following key points:

- Use a least-privileged account to connect to SQL.
- Credentials are passed over the wire so they must be secured.
- The SQL connection string (which contains credentials) must be secured.

Connection string types

If you connect to a SQL Server database using credentials (user name and password) then your connection string looks like this:

```
SqlConnectionString = "Server=YourServer;
                      Database=YourDatabase;
                      uid=YourUserName;pwd=YourStrongPassword;"
```

If you need to connect to a specific instance of SQL Server (a feature available only in SQL Server 2000 or later) installed on the same computer then your connection string looks like this:

```
SqlConnectionString = "Server=YourServer\Instance;
                      Database=YourDatabase;uid=YourUserName;
                      pwd=YourStrongPassword;"
```

If you want to connect to SQL Server using your network credentials, use the Integrated Security attribute (or **Trusted Connection** attribute) and omit the username and password:

```
SqlConnectionString = "Server=YourServer;
                       Database=YourDatabase;
                       Integrated Security=SSPI;"
```

- or -

```
SqlConnectionString = "Server=YourServer;
                      Database=YourDatabase;
                      Trusted_Connection=Yes;"
```

If you are connecting to an Oracle database by using explicit credentials (user name and password) then your connection string looks like this:

```
SqlConnectionString = "Provider=MSDAORA;Data Source=YourDatabaseAlias;
                      User ID=YourUserName;Password=YourPassword;"
```

More information

For more information about using Universal Data Link (UDL) files for your connection, see article Q308426, HOW TO: Use Data Link Files with the OleDbConnection Object in Visual C# .NET, in the Microsoft Knowledge Base.

Choosing a SQL account for your connections

Don't use the built-in **sa** or **db_owner** accounts for data access. Instead, use least-privileged accounts with a strong password.

Avoid the following connection string:

```
SqlConnectionString = "Server=YourServer\Instance;
                       Database=YourDatabase; uid=sa; pwd=;"
```

Use least-privileged accounts with a strong password, for example:

```
SqlConnectionString= "Server=YourServer\Instance;
                      Database=YourDatabase;
                      uid=YourStrongAccount;
```

```
pwd=YourStrongPassword;"
```

Note that this does not address the issue of storing credentials in plain text in your Web.config files. All you've done so far is limit the scope of damage possible in the event of a compromise, by using a least-privileged account. To further raise the security bar, you should encrypt the credentials.

Note If you selected a case-sensitive sort order when you installed SQL Server, your login ID is also case-sensitive.

Passing credentials over the network

When you connect to SQL Server with SQL authentication, the user name and password are sent across the network in clear text. This can represent a significant security concern. For more information about how to secure the channel between an application or Web server and database server, see Secure Communication later in this chapter.

Securing SQL connection strings

User names and passwords should not be stored in clear text in configuration files. For details about how to store connection strings securely, see "Storing Database Connection Strings" later in this chapter.

Authenticating Against Non-SQL Server Databases

The typical issues you may encounter when connecting to non-SQL databases are similar to scenarios where you need to use SQL authentication. You may need to supply explicit credentials if the target resources do not support Windows authentication. To secure this type of scenario, you must store the connection string securely and you must also secure the communication over the network (to prevent interception of credentials).

More information

- For more information about storing database connection strings, see Storing Database Connection Strings Securely later in this chapter.
- For more information about securing the channel to the database server, see Secure Communication later in this chapter.

Authorization

SQL Server provides a number of role-based approaches for authorization. These revolve around the following thee types of roles supported by SQL Server:

- **User-defined Database Roles**. These are used to group together users who have the same security privileges within the database. You add Windows user or group accounts to user database roles and establish permissions on individual database objects (stored procedures, tables, views, and so on) using the roles.
- **Application Roles**. These are similar to user database roles in that they are used when establishing object permissions. However, unlike user database roles, they do not contain users or groups. Instead, they must are activated by an application using a built-in stored procedure. Once active, the permissions granted to the role determine the data access capabilities of the application.

  Application roles allow database administrators to grant selected applications access to specified database objects. This is in contrast to granting permissions to users.

- **Fixed Database Roles**. SQL Server also provides fixed server roles such as db_datareader and db_datawriter. These built-in roles are present in all databases and can be used to quickly give a user read specific (and other commonly used) sets of permissions within the database.

For more information about these various role types (and also fixed server roles which are similar to fixed database roles but apply at the server level instead of the database level), see SQL Server Books Online.

Using Multiple Database Roles

If your application has multiple categories of users, and the users within each category require the same permissions within the database, your application requires multiple roles.

Each role requires a different set of permissions within the database. For example, members of an Internet User role may require read-only permissions to the majority of tables within a database, while members of an Administrator or Operator role may require read/write permissions.

Options

To accommodate these scenarios, you have two main options for role-based authorization within SQL Server:

- **User-defined SQL Server Database Roles**. These are used to assign permissions to database objects for groups of users who have the same security permissions within the database.

  When you use user-defined database roles, you check at the gate, map users to roles, (for example, in an ASP.NET Web application or in a middle-tier serviced component in an Enterprise Services server application) and use multiple identities to connect to the database, each of which maps to a user-defined database role.

- **SQL Application Roles**. These are similar to user-defined database roles in that they are used when you assign permissions to database objects. However, unlike user-defined database roles, they do not contain members and are activated from individual applications by using a built-in stored procedure.

  When you use application roles, you check at the gate, map users to roles, connect to the database using a single, trusted, service identity, and activate the appropriate SQL application role.

User-Defined database roles

If you elect to use user-defined database roles, you must:

- Create multiple service accounts to use for database access.
- Map each account to a user-defined database role.
- Establish the necessary database permissions for each role within the database.
- Authorize users within your application (ASP.NET Web application, Web service, or middle tier component) and then use application logic within your data access layer to determine which account to connect to the database with. This is based on the role-membership of the caller.

  Declaratively, you can configure individual methods to allow only those users that belong to a set of roles. You then add imperative role-checks within method code to determine precise role membership, which determines the connection to use.

Figure 12.4 illustrates this approach.

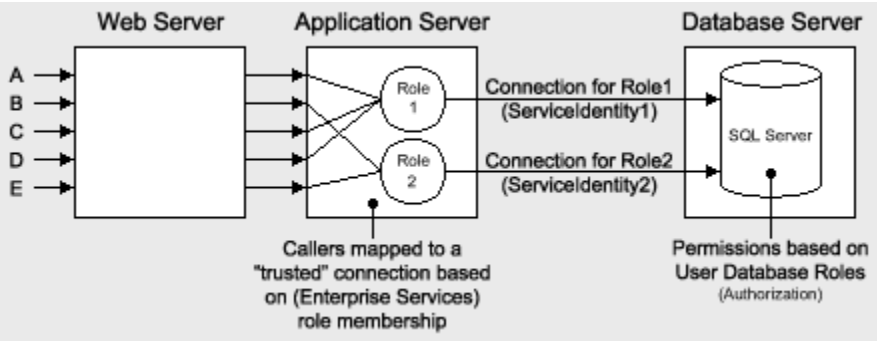

Figure 12.4. Connecting to SQL Server using multiple SQL user database roles

To use the preferred Windows authentication for this scenario, you develop code (using the **LogonUser** API) in an out of process serviced component to impersonate one of a set of Windows identities.

With SQL authentication, you use a different connection string (containing different user names and passwords) depending upon role-based logic within your application.

More information

For more information about securely storing database connection strings, see Storing Database Connection Strings Securely later in this chapter.

Application roles

With SQL application roles, you must:

- Create a single service account to use for database access (this may be the process account used to run the ASP.NET worker process, or an Enterprise Services application).
- Create a set of SQL application roles within the database.
- Establish the necessary database permissions for each role within the database.
- Authorize users within your application (ASP.NET Web application, Web service or middle tier component), and use application logic within your data access layer to determine which application role to activate within the database. This is based on the role-membership of the caller.

Figure 12.5 illustrates this approach.

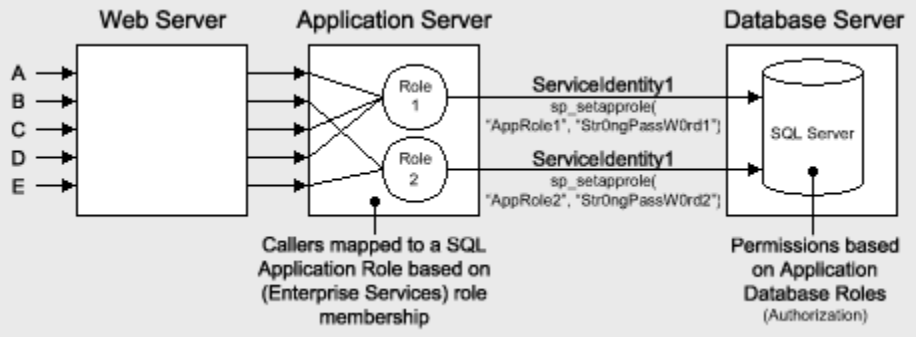

Figure 12.5. Using multiple SQL application roles

In Figure 12.5, the identity **ServiceIdentity1** that is used to access the database is usually obtained from the ASP.NET worker process or from an Enterprise Services server application process identity.

With this approach, the same service identity (and therefore the same connection) is used to connect to SQL Server. SQL application roles are activated with the **sp_setapprole** built-in stored procedure, based on the role membership of the caller. This stored procedure requires the role name and a password.

If you use this approach, you must securely store the role name and password credentials. For further advice and secret storage techniques, see Storing Database Connection Strings Securely later in this chapter.

Limitations of SQL application roles

The following are the key points that you must be aware of before you choose to use SQL application roles:

- You need to manage credentials for the SQL application roles. You must call the **sp_setapprole** stored procedure passing a role name and password for each connection. If you are activating a SQL application role from managed code then having a clear text password embedded in the assembly is not safe.
- SQL application role credentials are passed to the database in clear text. You should secure them on the network by using IPSec or SSL between the application server and database server.
- After a SQL application role is activated on a connection it cannot be deactivated. It remains active until the connection closes. Also, you cannot switch between two or more roles on the same connection.
- Use SQL application roles only when your application uses a single, fixed identity to connect to the database. In other words, use them only when your application uses the trusted subsystem model.

If the security context of the connection changes (as it would if the original caller's context were use to connect to the database), then SQL application roles do not work in conjunction with connection pooling.

For more information, see article Q229564, PRB: SQL Application Role Errors with OLE DB Resource Pooling, in the Microsoft Knowledge Base.

Secure Communication

In most application scenarios you need to secure the communication link between your application server and database. You need to be able to guarantee:

- **Message Confidentiality**. The data must be encrypted to ensure that it remains private.
- **Message Integrity**. The data must be signed to ensure that it remains unaltered.

In some scenarios, all of the data passed between application server and database server must be secured, while in other scenarios, selected items of data sent over specific connections must be secured. For example:

- In an intranet Human Resources application, some of the employee details passed between client and the database server are sensitive.
- In Internet scenarios, such as secure banking applications, all of the data passed between the application server and database server must be secured.
- If you are using SQL authentication, you should also secure the communication link to ensure that user names and passwords can not be compromised with network monitoring software.

The Options

You have two options for securing the network link between an application server and database server:

- IPSec
- SSL (using a server certificate on the SQL Server computer)

Note You must be running SQL Server 2000 to support the use of SSL. Earlier versions do not support it. The client must have the SQL Server 2000 client libraries installed.

Choosing an Approach

Whether or not you should use IPSec or SSL depends on a number of primarily environmental factors, such as firewall considerations, operating system and database versions, and so on.

Note IPSec is not intended as a replacement for application-level security. Today it is used as a defense in depth mechanism, or to secure insecure applications without changing them, and to secure non-TLS (for example, SSL) protocols from network-wire attacks.

More information

- For more information about configuring IPSec, see How To: Use IPSec to Provide Secure Communication Between Two Servers in the Reference section of this guide.
- For more information about configuring SSL, see How To: Use SSL to Secure Communication with SQL Server 2000 in the Reference section of this guide.
- For more information about SSL and IPSec in general, see Chapter 4, Secure Communication.

Connecting with Least Privilege

Connecting to the database with least privilege means that the connection you establish only has the minimum privileges that you need within the database. Simply put, you don't connect to your database using the **sa** or database owner accounts. Ideally, if the current user is not authorized to add or update records, then the

corresponding account used for their connection (which may be aggregated to an identity that represents a particular role) cannot add or update records in the database.

When you connect to SQL Server, your approach needs to support the necessary granularity that your database authorization requires. You need to consider what the database trusts. It can trust:

- The application
- Application-defined roles
- The original caller

The Database Trusts the Application

Consider a finance application that you authorize to use your database. The finance application is responsible for managing user authentication and authorizing access. In this case, you can manage your connections through a single trusted account (which corresponds to either a SQL login or a Windows account mapped to a SQL login). If you're using Windows authentication, this would typically mean allowing the process identity of the calling application (such as the ASP.NET worker process, or an Enterprise Services server application identity) to access the database.

From an authorization standpoint, this approach is very coarse-grained, because the connection runs as an identity that has access to all database objects and resources needed by the application. The benefits of this approach are that you can use connection pooling and you simplify administration because you are authorizing a single account. The downside is that all of your users run with the same connection privileges.

The Database Trusts Different Roles

You can use pools of separate, trusted connections to the database that correspond to the roles defined by your application, for example, one connection that is for tellers, another for managers, and so on.

These connections may or may not use Windows authentication. The advantage of Windows authentication is that it handles credential management and doesn't send the credentials over the network. However, while Windows authentication is possible at the process or application level (as when you use a single connection to the database), there are additional challenges presented by the fact you need multiple identities (one per role).

Many applications use the **LogonUser** API to establish a Windows access token. The problem with this approach is two-fold:

- You now have a credential management issue (your application has to securely store the account user name and password).
- The **LogonUser** API requires that the calling process account have the "Act as part of the operating system" privilege. This means that you are forced to give the ASP.NET process account this privilege, which is not recommended. An alternative is to use SQL Authentication, but then you need to protect the credentials on the server and over the network.

  Note This **LogonUser** restriction is lifted in Windows Server 2003.

The Database Trusts the Original Caller

In this case, you need to flow the original caller through multiple tiers to the database. This means that your clients need network credentials to be able to hop from one computer to the next. This requires Kerberos delegation.

Although this solution provides a fine-grained level of authorization within the database, because you know the identity of the original caller and can establish per user permissions on database objects, it impacts application performance and scalability. Connection pooling (although still enabled) becomes ineffective.

Creating a Least Privilege Database Account

The following steps are provided as a simple example to show you how to create a least privilege database account. While most database administrators are already familiar with these steps, many developers may not be and resort to using the **sa** or database owner account to force their applications to work.

This can create difficulties when moving from a development environment, to a test environment, and then to a production environment because the application moves from an environment that's wide open into a more tightly controlled setting, which prevents the application from functioning correctly.

You start by creating a SQL login for either a SQL account or a Windows account (user or group). You then add that login to a database user role and assign permissions to that role.

To set up a data access account for SQL

1. Create a new user account and add the account to a Windows group. If you are managing multiple users, you would use a group. If you are dealing with a single application account (such as a duplicated ASP.NET process account), you may choose not to add the account to a Windows group.
2. Create a SQL Server login for the user/group.
   a. Start **Enterprise Manager**, locate your database server, and then expand the **Security** folder.
   b. Right-click **Logins**, and then click **New Login**.
   c. Enter the Windows group name into the **Name** field, and then click **OK** to close the **SQL Server Login Properties** dialog box.
3. Create a new database user in the database of interest that is mapped to the SQL server login.
   a. Use Enterprise Manager and expand the Databases folder, and then expand the required database for which the login requires access.
   b. Right-click **Users**, and then click **New Database User**.
   c. Select the previously created Login name.
   d. Specify a user name.
   e. Configure permissions as discussed below.
4. Grant the database user **Select** permissions on the tables that need to be accessed and **Execute** permissions on any relevant stored procedures.

   Note If the stored procedure and the table are owned by the same person, and access the table only through the stored procedure (and do not need to access the table directly), it is sufficient to grant execute permissions on the stored procedure alone. This is because of the concept of ownership chaining. For more information, see SQL Server Books online.

5. If you want the user account to have access to all the views and tables in the database, add them to the **db_datareader** role.

Storing Database Connection Strings Securely

There are a number of possible locations and approaches for storing database connection strings, each with varying degrees of security and configuration flexibility.

The Options

The following list represents the main options for storing connection strings:

- Encrypted with DPAPI
- Clear text in Web.config or Machine.config
- UDL files
- Custom text files
- Registry
- COM+ catalog

Using DPAPI

Windows 2000 and later operating systems provide the Win32® Data Protection API (DPAPI) for encrypting and decrypting data. DPAPI is part of the Cryptography API (Crypto API) and is implemented in Crypt32.dll. It consists of two methods—**CryptProtectData** and **CryptUnprotectData**.

DPAPI is particularly useful in that it can eliminate the key management problem exposed to applications that use cryptography. While encryption ensures the data is secure, you must take additional steps to ensure the security of the key. DPAPI uses the password of the user account associated with the code that calls the DPAPI functions in order to derive the encryption key. As a result the operating system (and not the application) manages the key.

Why not LSA?

Many applications use the Local Security Authority (LSA) to store secrets. DPAPI has the following advantages over the LSA approach:

- To use the LSA, a process requires administrative privileges. This is a security concern because it greatly increases the potential damage that can be done by an attacker who manages to compromise the process.
- The LSA provides only a limited number of slots for secret storage, many of which are already used by the system.

Machine store vs. user store

DPAPI can work with either the machine store or user store (which requires a loaded user profile). DPAPI defaults to the user store, although you can specify that the machine store be used by passing the CRYPTPROTECT_LOCAL_MACHINE flag to the DPAPI functions.

The user profile approach affords an additional layer of security because it limits who can access the secret. Only the user who encrypts the data can decrypt the data. However, use of the user profile requires additional development effort when DPAPI is used from an ASP.NET Web application because you need to take explicit steps to load and unload a user profile (ASP.NET does not automatically load a user profile).

The machine store approach is easier to develop because it does not require user profile management. However, unless an additional entropy parameter is used, it is less secure because any user on the computer can decrypt data. (Entropy is a random value designed to make deciphering the secret more difficult). The problem with using an additional entropy parameter is that this must be securely stored by the application, which presents another key management issue.

Note If you use DPAPI with the machine store, the encrypted string is specific to a given computer and therefore you must generate the encrypted data on every computer. Do not copy the encrypted data across computers in a farm or cluster.
If you use DPAPI with the user store, you can decrypt the data on any computer with a roaming user profile.

DPAPI implementation solutions

This section presents two implementation solutions that show you how to use DPAPI from an ASP.NET Web application to secure a connection string (or a secret of any type). The implementation solutions described in this section are:

- **Using DPAPI from Enterprise Services**. This solution allows you to use DPAPI with the user store.
- **Using DPAPI directly from ASP.NET**. This solution allows you to use DPAPI with the machine store, which makes the solution easier to develop as DPAPI can be called directly from an ASP.NET Web application.

Using DPAPI from Enterprise Services

An ASP.NET Web application can't call DPAPI and use the user store because this requires a loaded user profile. The ASPNET account usually used to run Web applications is a non-interactive account and as such does not have a user profile. Furthermore, if the ASP.NET application is impersonating, the Web application thread runs as the currently authenticated user, which can vary from one request to the next.

This presents the following issues for an ASP.NET Web application that wants to use DPAPI:

- Calls to DPAPI from an ASP.NET application running under the default ASPNET account will fail. This is because the ASPNET account does not have a user profile, as it is not used for interactive logons.
- If an ASP.NET Web application is configured to impersonate its callers, the ASP.NET application thread has an associated thread impersonation token. The logon session associated with this impersonation token is a network logon session (used on the server to represent the caller). Network logon sessions do not result in user profiles being loaded.

To overcome this issue, you can create a serviced component (within an out-of-process Enterprise Services (COM+) server application) to call DPAPI. You can ensure that the account used to run the component has a user profile and you can use a Win32 service to automatically load the profile.

Note It is possible to avoid the use of a Win32 service by placing calls to Win32 profile management functions (**LoadUserProfile** and **UnloadUserProfile**) within the serviced component.
There are two drawbacks to this approach. First, calls to these APIs on a per-request basis would severely impact performance. Second, these APIs require that the calling code have administrative privileges on the local computer, which defeats the principle of least privilege for the Enterprise Services process account.

Figure 12.6 shows the Enterprise Services DPAPI solution.

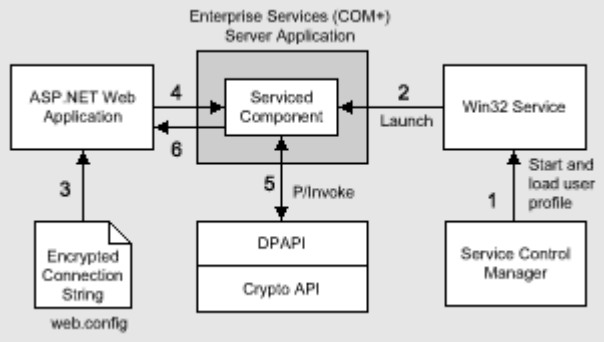

Figure 12.6. The ASP.NET Web application uses a COM+ server application to interact with DPAPI

In Figure 12.6, the runtime sequence of events is as follows:

1. The Windows service control manager starts the Win32 service and automatically loads the user profile associated with the account under which the service runs. The same Windows account is used to run the Enterprise Services application.
2. The Win32 service calls a launch method on the serviced component that starts the Enterprise Services application and loads the serviced component.
3. The Web application retrieves the encrypted string from the Web.config file.

   You can store the encrypted string by using an <**appSettings**> element within Web.config as shown below. This element supports arbitrary key-value pairs.

```
<configuration>
 <appSettings>
  <add key="SqlConnString"
       value="AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAABcqc/xCHxki3" />
 </appSettings>
</configuration>
```

   You can retrieve the encrypted string with the following line of code:

```
string connString = ConfigurationSettings.AppSettings["SqlConnString"];
```

> **Note** You can use Web.config or Machine.config to store encrypted connection strings. Machine.config is preferred as it is in a system directory outside of a virtual directory. This is discussed further in the next section, "Using Web.config and Machine.config."

4. The application calls a method on the serviced component to decrypt the connection string.
5. The serviced component interacts with DPAPI using P/Invoke to call the Win32 DPAPI functions.
6. The decrypted string is returned to the Web application.

Note To store encrypted connection strings in the Web.config file in the first place, write a utility application that takes the connection strings and calls the serviced component's **EncryptData** method to obtain the encrypted string. It is essential that you run the utility application while logged on with the same account that you use to run the Enterprise Services server application.

Using DPAPI directly from ASP.NET

If you use the machine store (and call the DPAPI functions with the CRYPTPROTECT_LOCAL_MACHINE flag) you can call the DPAPI functions directly from an ASP.NET Web application (because you don't need a user profile).

However, because you are using the machine store, any Windows account that can log on to the computer has access to the secret. A mitigating approach is to add entropy but this requires additional key management.

As alternatives to using entropy with the machine store, consider the following options:

- Use Windows ACLs to restrict access to the encrypted data (whether the data is stored in the file system or registry).
- Consider hard-coding the entropy parameter into your application to avoid the key management issue.

More information

- For more information about creating a DPAPI library for use with .NET Web applications, see How To: Create a DPAPI Library in the Reference section of this guide.
- For a detailed implementation walkthrough that shows you how to use DPAPI directly from ASP.NET, see How To: Use DPAPI (Machine Store) from ASP.NET in the Reference section of this guide.
- For a detailed implementation walkthrough that shows you how to use DPAPI from Enterprise Services, see How To: Use DPAPI (User Store) from ASP.NET with Enterprise Services in the Reference section of this guide.
- For more information about Windows Data Protection with DPAPI, see the MSDN article, Windows Data Protection.

Using Web.config and Machine.config

Storing plain text passwords in Web.config is not recommended. By default, the **HttpForbiddenHandler** protects the file from being downloading and viewed by malicious users. However, users who have access directly to the folders that contain the configuration files can still see the user name and password.

Machine.config is considered a more secure storage location than Web.config because it is located in a system directory (with ACLs) outside of a Web application's virtual directory. Always lock down Machine.config with ACLs.

More information

For more information about securing Machine.config, see Chapter 8, ASP.NET Security.

Using UDL Files

The OLE DB .NET Data Provider supports UDL file names in its connection string. To reference a UDL file, use "File Name=name.udl" within the connection string.

Important This option is only available if you use the OLE DB .NET Data Provider to connect to the database. The SQL Server .NET Data Provider does not use UDL files.

It is not recommended to store UDL files in a virtual directory along with other application files. You should store them outside the Web application's virtual directory hierarchy and then secure the file or the folder containing the file with Windows ACLs. You should also consider storing UDL files on a separate logical volume from the operating system to protect against possible file canonicalization and directory traversal bugs.

ACL granularity

UDL files (or indeed any text file) offer added granularity when you apply ACLs in comparison to Machine.config. The default ACLs associated with Machine.config grant access to a wide variety of local and remote users. For example, Machine.config has the following default ACLs:

```
MachineName\ASPNET:R

BUILTIN\Users:R

BUILTIN\Power Users:C

BUILTIN\Administrators:F

NT AUTHORITY\SYSTEM:F
```

By contrast, you can lock down your own application's UDL file much further. For example, you can restrict access to Administrators, the System account, and the ASP.NET process account (which requires read access) as shown below.

```
BUILTIN\Administrators:F

MachineName\ASPNET:R

NT AUTHORITY\SYSTEM:F
```

Note Because UDL files can be modified externally to any ADO.NET client application, connection strings that contain references to UDL files are parsed every time the connection is opened. This can impact performance and it is therefore recommended, for best performance, that you use a static connection string that does not include a UDL file.

To create a new UDL file

1. Open the folder in which you want to create the UDL file.
2. Right-click within the folder, point to **New,** and then click **Text Document**.
3. Supply a file name with a .udl file extension.
4. Double-click the new file to display the **UDL Properties** dialog box.

More information

For more information about using UDL files from Microsoft C#® development tool programs, see article Q308426, HOW TO: Use Data Link Files with OleDbConnection Object in VC#, in the Microsoft Knowledge Base.

Using Custom Text Files

Many applications use custom text files to store connection strings. If you do adopt this approach consider the following recommendations:

- Store custom files outside of your application's virtual directory hierarchy.
- Consider storing files on a separate logical volume from the operating system to protect against possible file canonicalization and directory traversal bugs.
- Protect the file with a restricted ACL that grants read access to your application's process account.
- Avoid storing the connection string in clear text in the file. Instead, consider using DPAPI to store an encrypted string.

Using the Registry

You can use a custom key in the Windows registry to store the connection string. This information stored can either be stored in the HKEY_LOCAL_MACHINE (HKLM) or HKEY_CURRENT_USER (HKCU) registry hive. For process identities, such as the ASPNET account, that do not have user profiles, the information must be stored in HKLM in order to allow ASP.NET code to retrieve it.

If you do use this approach, you should:

- Use ACLs to protect the registry key using Regedt32.exe.
- Encrypt the data prior to storage.

More information

For more information about encrypting data for storage in the registry, see How To: Store an Encrypted Connection String in the Registry in the Reference section of this guide.

Using the COM+ Catalog

If your Web application includes serviced components, you can store connection strings in the COM+ catalog as constructor strings. These are easily administered (by using the Component Services tool) and are easily retrieved by component code. Enterprise Services calls an object's **Construct** method immediately after instantiating the object, and passes the configured construction string.

The COM+ catalog doesn't provide a high degree of security, because the information is not encrypted; however, it raises the security bar in comparison to configuration files because of the additional process hop.

To prevent access to the catalog through the Component Services tool, include only the desired list of users in the **Administrator** and **Reader** roles in the **System** application.

The following example shows how to retrieve an object constructor string from a serviced component.

```
[ConstructionEnabled(Default="Default Connection String")]

public class YourClass : ServicedComponent

{

  private string _ConnectionString;

  override protected void Construct(string s)

  {

    _ConnectionString = s;

  }

}
```

For added security, you can add code to encrypt the construction string prior to storage and decrypt it within the serviced component.

More information

- For more information on using connection strings, see article Q271284, HOWTO: Access COM+ Object Constructor String in a VB Component, in the Microsoft Knowledge Base.
- For a complete code sample provided by the .NET Framework SDK, see the object constructor sample located in \Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Samples\Technologies\ComponentServices\ObjectConstruction.

Authenticating Users Against a Database

If you are building an application that needs to validate user credentials against a database store, consider the following points:

- Store one-way password hashes (with a random salt value).
- Avoid SQL injection when validating user credentials.

Store One-way Password Hashes (with Salt)

Web applications that use Forms authentication often need to store user credentials (including passwords) in a database. For security reasons, you should not store passwords (clear text or encrypted) in the database.

You should avoid storing encrypted passwords because it raises key management issues—you can secure the password with encryption, but you then have to consider how to store the encryption key. If the key becomes compromised, an attacker can decrypt all the passwords within your data store.

The preferred approach is to:

- **Store a one way hash of the password**. Re-compute the hash when the password needs to be validated.
- **Combine the password hash with a salt value (a cryptographically strong random number)**. By combining the salt with the password hash, you mitigate the threat associated with dictionary attacks.

Creating a salt value

The following code shows how to generate a salt value by using random number generation functionality provided by the **RNGCryptoServiceProvider** class within the **System.Security.Cryptography** namespace.

```
public static string CreateSalt(int size)

{

  RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

  byte[] buff = new byte[size];

  rng.GetBytes(buff);

  return Convert.ToBase64String(buff);

}
```

Creating a hash value (with salt)

The following code fragment shows how to generate a hash value from a supplied password and salt value.

```
public static string CreatePasswordHash(string pwd, string salt)
{
  string saltAndPwd = string.Concat(pwd, salt);
  string hashedPwd =
        FormsAuthentication.HashPasswordForStoringInConfigFile(
                                              saltAndPwd, "SHA1");
  return hashedPwd;
}
```

More information

For the full implementation details of this approach, see How To: Use Forms Authentication with SQL Server 2000 in the Reference section of this guide.

SQL Injection Attacks

If you're using Forms authentication against a SQL database, you should take the precautions discussed in this section to avoid SQL injection attacks. SQL injection is the act of passing additional (malicious) SQL code into an application which is typically appended to the legitimate SQL code contained within the application. All SQL databases are susceptible to SQL injection to varying degrees, but the focus in this chapter is on SQL Server

You should pay particular attention to the potential for SQL injection attacks when you process user input that forms part of a SQL command. If your authentication scheme is based on validating users against a SQL database, for example, if you're using Forms authentication against SQL Server, then you must guard against SQL injection attacks.

If you build SQL strings using unfiltered input, your application may be subject to malicious user input (remember, never trust user input). The risk is that when you insert user input into a string that becomes an executable statement, a malicious user can append SQL commands to your intended SQL statements by using escape characters.

The code fragments in the following sections use the Pubs database that is supplied with SQL Server to illustrate examples of SQL injection.

The Problem

Your application may be susceptible to SQL injection attacks when you incorporate user input or other unknown data into database queries. For example, both of the following code fragments are susceptible to attack.

- You build SQL statements with unfiltered user input.

```
SqlDataAdapter myCommand = new SqlDataAdapter(
                "SELECT au_lname, au_fname FROM authors WHERE au_id = '" +
                Login.Text + "'", myConnection);
```

- You call a stored procedure by building a single string that incorporates unfiltered user input.

```
•       SqlDataAdapter myCommand = new SqlDataAdapter("LoginStoredProcedure '" +
•                                       Login.Text + "'", myConnection);
```

Anatomy of a SQL Script Injection Attack

When you accept unfiltered user input values (as shown above) in your application, a malicious user can use escape characters to append their own commands.

Consider a SQL query that expects the user's input to be in the form of a Social Security Number, such as 172-32-xxxx, which results in a query like this:

```
SELECT au_lname, au_fname FROM authors WHERE au_id = '172-32-xxxx'
```

A malicious user can enter the following text into your application's input field (for example a text box control).

```
' ; INSERT INTO jobs (job_desc, min_lvl, max_lvl) VALUES ('Important Job',
25, 100)  -
```

In this example, an INSERT statement is injected (but any statement that is permitted for the account that's used to connect to SQL Server could be executed). The code can be especially damaging if the account is a member of the **sysadmin** role (this allows shell commands using **xp_cmdshell**) and SQL Server is running under a domain account with access to other network resources.

The command above results in the following combined SQL string:

```
SELECT au_lname, au_fname FROM authors WHERE au_id = '';INSERT INTO

jobs (job_desc, min_lvl, max_lvl) VALUES ('Important Job', 25, 100)  --
```

In this case, the ' (single quotation mark) character that starts the rogue input terminates the current string literal in your SQL statement. It closes the current statement only if the following parsed token doesn't make sense as a continuation of the current statement, but does make sense as the start of a new statement.

```
SELECT au_lname, au_fname FROM authors WHERE au_id = ' '
```

The ; (semicolon) character tells SQL that you're starting a new statement, which is then followed by the malicious SQL code:

```
; INSERT INTO jobs (job_desc, min_lvl, max_lvl) VALUES ('Important

Job', 25, 100)
```

Note The semicolon is not necessarily required to separate SQL statements. This is vendor/implementation dependent, but SQL Server does not require them. For example, SQL Server will parse the following as two separate statements:

```
SELECT * FROM MyTable DELETE FROM MyTable
```

Finally, the -- (double dash) sequence of characters is a SQL comment that tells SQL to ignore the rest of the text, which in this case, ignores the closing ' (single quote) character (which would otherwise cause a SQL parser error).

The full text that SQL executes as a result of the statement shown above is:

```
SELECT au_lname, au_fname FROM authors WHERE au_id = '' ; INSERT INTO

jobs (job_desc, min_lvl, max_lvl) VALUES ('Important Job', 25, 100) --'
```

The solution

The following approaches can be used to call SQL safely from your application.

- Use the **Parameters** collection when building your SQL statements.

```
SqlDataAdapter myCommand = new SqlDataAdapter(
        "SELECT au_lname, au_fname FROM Authors WHERE au_id=
@au_id",
        myConnection);

SqlParameter parm = myCommand.SelectCommand.Parameters.Add(
                                            "@au_id",
                                            SqlDbType.VarChar, 11);
parm.Value= Login.Text;
```

- Use the **Parameters** collection when you call a stored procedure.

```
// AuthorLogin is a stored procedure that accepts a parameter
// named Login
SqlDataAdapter myCommand = new SqlDataAdapter("AuthorLogin",
myConnection);
myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
SqlParameter parm = myCommand.SelectCommand.Parameters.Add(
                                "@LoginId", SqlDbType.VarChar,11);
parm.Value=Login.Text;
```

If you use the **Parameters** collection, no matter what a malicious user includes as input, the input is treated as a literal. An additional benefit of using the **Parameters** collection is that you can enforce type and length checks. Values outside of the range trigger an exception. This is a healthy example of defense in depth.

- Filter user input for SQL characters. The following method shows how to ensure that any string literal used in a simple SQL comparison statement (equal to, less than, greater than) is safe. It does this by ensuring that any apostrophe used in the string is escaped with an additional apostrophe. Within a SQL string literal, two consecutive apostrophes are treated as an instance of the apostrophe character within the string rather than as delimiters.

```
•       private string SafeSqlLiteral(string inputSQL)
•       {
•         return inputSQL.Replace("'", "''");
•       }
•       ...
•       string safeSQL = SafeSqlLiteral(Login.Text);
•       SqlDataAdapter myCommand = new SqlDataAdapter(
•              "SELECT au_lname, au_fname FROM authors WHERE au_id = '" +
•              safeSQL + "'", myConnection);
```

Additional best practices

The following are some additional measures you can take to limit the chance of exploit, as well as limit the scope of potential damage:

- Prevent invalid input at the gate (the front-end application) by limiting the size and type of input. By limiting the size and type of input, you significantly reduce the potential for damage. For example, if your database lookup field is eleven characters long and comprised entirely of numeric characters, enforce it.
- Run SQL code with a least privileged account. This significantly reduces the potential damage that can be done.

  For example, if a user were to inject SQL to DROP a table from the database, but the SQL connection used an account that didn't have appropriate permissions, the SQL code would fail. This is another reason not to use the **sa** account or database owner account for your application's SQL connections.

- When an exception occurs in your SQL code, do not expose the SQL errors raised by the database to the end user. Log error information and show only user friendly information. This prevents exposing unnecessary detail that could help an attacker.

Protecting Pattern Matching Statements

If input is to be used within string literals in a 'LIKE' clause, characters other than apostrophe also take on special meaning for pattern matching.

For example, in a LIKE clause the % character means "match zero or more characters." In order to treat such characters in the input as literal characters without special meaning, they also need to be escaped. If they are not handled specially, the query can return incorrect results; a non-escaped pattern matching character at or near the beginning of the string could also defeat indexing.

For SQL Server, the following method should be used to ensure valid input:

```
private string SafeSqlLikeClauseLiteral(string inputSQL)
{
  // Make the following replacements:
  // '  becomes  ''
```

```
  // [  becomes  [[]

  // %  becomes  [%]

  // _  becomes  [_]


  string s = inputSQL;

  s = inputSQL.Replace("'", "''");

  s = s.Replace("[", "[[]");

  s = s.Replace("%", "[%]");

  s = s.Replace("_", "[_]");

  return s;

}
```

Auditing

Auditing of logons is not on by default within SQL Server. You can configure this either through SQL Server Enterprise Manager or in the registry. The dialog box in Figure 12.7 shows auditing enabled for both the success and failure of user logons.

Log entries are written to SQL log files which are by default located in C:\Program Files\Microsoft SQL Server\MSSQL\LOG. You can use any text reader, such as Notepad, to view them.

Figure 12.7. SQL Server Properties dialog with Audit level settings

You can also enable SQL Server auditing in the registry. To enable SQL Server auditing, create the following **AuditLevel** key within the registry and set its value to one of the REG_DWORD values specified below.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\AuditLevel
```

You can choose from one of the following values, which allow you to capture the level of detail you want:

3—captures both success and failed login attempts.

2—captures only failed login attempts.

1—captures only success login attempts.

0—captures no logins.

It is recommended that you turn on failed login auditing because this is a way to determine if someone is attempting a brute attack into SQL Server. The performance impacts of logging failed audit attempts are minimal unless you are being attacked, in which case you need to know anyway.

You can also script against SQL Database Management Objects (DMO). The following code fragment shows some sample VBScript code.

```
Sub SetAuditLevel(Server As String, NewAuditLevel As SQLDMO_AUDIT_TYPE)
```

```
    Dim objServer As New SQLServer2

    objServer.LoginSecure = True     'Use integrated security

    objServer.Connect Server        'Connect to the target SQL Server

    'Set the audit level

    objServer.IntegratedSecurity.AuditLevel = NewAuditLevel

    Set objServer = Nothing

End Sub
```

From SQL Server Books online, the members of the enumerated type, SQLDMO_AUDIT_TYPE are:

```
SQLDMOAudit_All      3  Log all authentication attempts regardless of success
                        or failure

SQLDMOAudit_Failure  2    Log failed authentication

SQLDMOAudit_Success  1    Log successful authentication

SQLDMOAudit_None     0    Do not log authentication attempts
```

Process Identity for SQL Server

Run SQL Server using a least-privileged domain account. When you install SQL Server, you have the option of running the SQL Server service using the local SYSTEM account, or a specified account.

Don't use the SYSTEM account or an administrator account. Instead, use a least-privileged domain account. You do not need to grant this account any specific privileges, as the installation process (or SQL Server Enterprise Manager, if you are reconfiguring the SQL Service after installation) grants the specified account the necessary privileges.

Summary

The following is a summary that highlights the recommendation for data access in your .NET Web applications:

- Use Windows authentication to SQL Server when possible.
- Use accounts with least privilege in the database.
- Use least-privileged, local accounts for running ASP.NET/Enterprise Services when connecting to SQL Server.
- If you are using SQL authentication, take the following steps to improve security:
  - Use custom accounts with strong passwords.
  - Limit the permissions of each account within SQL Server using database roles.
  - Add ACLs to any files used to store connection strings.
  - Encrypt connection strings.
  - Consider DPAPI for credential storage.
- When you use Forms authentication against SQL, take precautions to avoid SQL injection attacks.

- Don't store user passwords in databases for user validation. Instead, store password hashes with a salt instead of clear text or encrypted passwords.
- Protect sensitive data sent over the network to and from SQL Server.
  - Windows authentication protects credentials, but not application data.
  - Use IPSec or SSL.

Troubleshooting Security Issues

J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy
Microsoft Corporation

November 2002

Applies to:
Microsoft® ASP.NET

See the Landing Page for the starting point and a complete overview of *Building Secure ASP.NET Applications*.

Summary: This section presents a set of troubleshooting tips, techniques and tools to help diagnose security related issues. (19 printed pages)

Contents

Process for Troubleshooting

The following approach has proven to be helpful for resolving security and security context related issues.

1. Start by describing the problem very clearly. Make sure you know precisely what is supposed to happen, what is actually happening, and most importantly, the detailed steps required to reproduce the problem.

2. Isolate the problem as accurately as you can. Try to determine at which stage during the processing of a request the problem occurs. Is it a client or server related issue? Does it appear to be a configuration or code related error? Try to isolate the problem by stripping away application layers. For example, consider building a simple console-based test client application to take the place of more complex client applications.

3. Analyze error messages and stack traces (if they are available). Always start by consulting the Windows event and security logs.

4. Check the Microsoft Knowledge Base to see if the problem has been documented as a Knowledge Base article.

5. Many security problems relate to the identity used to run code; these are not always the identities you imagine are running the code. Use the code samples presented in the Determining Identity subsection of the "ASP.NET" section in this chapter to retrieve and diagnose identity information. If the identities appear incorrect, check the configuration settings in web.config and machine.config and also check the IIS authentication settings for your application's virtual directory. Factors that can affect identity within an ASP.NET Web application include:

   - The <**processModel**> element in machine.config used to determine the process identity of the ASP.NET worker process (aspnet_wp.exe)
   - Authentication settings in IIS
   - Authentication settings in web.config
   - Impersonation settings in web.config

6. Even if it appears that the correct settings are being used and displayed, you may want to explicitly configure a web.config file for your application (in the application's virtual directory) to make sure it is not

inheriting settings from a higher-level application (perhaps from a web.config in a higher-level virtual directory) or from machine.config.

7. Use some of the troubleshooting tools listed in the Troubleshooting Tools section later in this chapter to capture additional diagnostics.

8. Attempt to reproduce the problem on another computer. This can help isolate environmental-related problems and can indicate whether or not the problem is in your application's code or configuration.

9. If your application is having problems accessing a remote resource, you may be running into impersonation/delegation related problems. Identify the security context being used for the remote resource access, and if you are using Windows authentication, verify that the account providing the context (for example, a process account), should be able to be authenticated by the remote computer.

10. Search newsgroups to see if the problem has already been reported. If not, post the problem to the newsgroup to see if anyone within the development community can provide assistance.

    The online newsgroup for ASP.NET is located at: http://communities.microsoft.com/newsgroups/default.asp?icp=mscom&slcid=US&newsgroup=microsoft.public.dotnet.framework.aspnet.

11. Call the Microsoft Support Center. For details, see the Microsoft Knowledge Base.

Searching for Implementation Solutions

If you have a specific issue and need to understand the best way to tackle the problem, use the following approach.

- Search in Chapters 5, 6, and 7 of this guide for your scenario or a similar scenarios.
- Consult the MSDN library documentation and samples.
- Refer to one of the many ASP.NET information Web sites, such as:
  - www.asp.net
  - www.gotdotnet.com
  - www.asptoday.com
- Search the Microsoft Knowledge Base for an appropriate How To article.
- Post questions to newsgroups.
- Call the Microsoft Support Center.

Troubleshooting Authentication Issues

The first step when troubleshooting authentication issues is to distinguish between IIS and ASP.NET authentication failure messages.

- If you are receiving an IIS error message you will not see an ASP.NET error code. Check the IIS authentication settings for your application's virtual directory.

  Create a simple HTML test page to remove ASP.NET from the solution.

- If you are receiving an ASP.NET error message, review the ASP.NET authentication settings within your application's web.config file.

IIS Authentication Issues

Because the authentication process starts with IIS, make sure IIS is configured correctly.

- Make sure a user is being authenticated. Consider enabling just Basic authentication and manually log in to ensure you know what principal is being authenticated. Log in with a user name of the form "domain\username".
- Restart IIS to ensure log on sessions aren't being cached. (Run IISReset.exe to restart IIS.)
- Close your browser between successive tests to ensure the browser isn't caching credentials.
- If you are using Integrated Windows authentication, check browser settings as described below.
  - Click **Tools** from the **Internet Options** menu and then click the **Advanced** tab. Select **Enable Integrated Windows Authentication (requires restart)**. Then restart the browser.
  - Click **Tools** from the **Internet Options** menu, and then click the **Security** tab. Select the appropriate Web content zone and click **Custom Level**. Within **User Authentication** ensure the Logon setting is set correctly for your application. You may want to select **Prompt for user name and password** to ensure that for each test you are providing explicit credentials and that nothing is being cached.
  - If the browser prompts you for credentials, this could mean you are currently logged into a domain that the server doesn't recognize (for example, you may be logged in as administrator on the local machine).
  - When you browse to an application on your local computer, your interactive logon token is used, as you are interactively logged onto the Web server.
- Test with a simple Web page that displays security context information. A sample page is provided later in this chapter.

  If this fails, enable auditing on the requested file and check the Security event log. You must also enable auditing using Group Policy (through either the Local Security Policy tool, or the Domain Security Policy tool). Examine the log for invalid usernames or invalid object access attempts.

  - If your Web application is having problems accessing a remote resource, enable auditing on the remote resource.
  - An invalid username and/or password usually means that the account used to run ASP.NET on your Web server is failing to be correctly authenticated at the remote computer. If you are attempting to access remote resources with the default ASPNET local account, check that you have duplicated the account (and password) on the remote computer.
  - If you see an error message that indicates that the login has failed for NT AUTHORITY\ANONYMOUS, this indicates that the identity on the Web server does not have any network credentials and is attempting to access the remote computer.

    Identify which account is being used by the Web application for remote resource access and confirm that it has network credentials. If the Web application is impersonating, this requires either Kerberos delegation (with suitably configured accounts) or Basic authentication at the Web server.

Using Windows Authentication

If the <**authentication**> element in your application's web.config is configured for Windows authentication, use the following code in your Web application to check whether anonymous access is being used (and the authenticated user is the anonymous Internet user account [IUSR_MACHINE]).

```
WindowsIdentity winId = HttpContext.Current.User.Identity as WindowsIdentity;

if (null != winId)

{

  Response.Write(winId.IsAnonymous.ToString());

}
```

Using Forms Authentication

Make sure that the cookie name specified in the <**forms**> element is being retrieved in the global.asax event handler correctly (**Application_AuthenticateRequest**). Also, make sure the cookie is being created. If the client is continuously sent back to the login page (specified by the **loginUrl** attribute on the <**forms**> element) this indicates that the cookie is not being created for some reason, or an authenticated identity is not being placed into the context (**HttpContext.User**)

Kerberos Troubleshooting

Use the following tools to help troubleshoot Kerberos-related authentication and delegation issues.

- **Kerbtray.exe**. This utility can be used to view the Kerberos tickets in the cache on the current computer. It is part of the Windows 2000 Resource Kit and can be downloaded from the Microsoft Download Center. Search for "Kerbtray.exe".
- **Klist.exe**. This is a command line tool similar to Kerbtray, but it also allows you to view and delete Kerberos tickets. Once again, it is part of the Windows 2000 Resource Kit and can be downloaded from the Microsoft Download Center. Search for "Klist.exe"
- **Setspn.exe**. This is a command-line tool that allows you to manage the Service Principal Names (SPN) directory property for an Active Directory service account. SPNs are used to locate a target principal name for running a service. It is part of the Windows 2000 Resource Kit and can be downloaded from the Microsoft Download Center. Search for "setspn.exe".

Troubleshooting Authorization Issues

Check Windows ACLs

If your application is having problems accessing a file or registry key (or any securable Windows object protected with ACLs), check the ACLs to ensure that the Web application identity has at least read access.

Check Identity

Also make sure you know which identity is being used for resource access by the ASP.NET Web application. This is likely to be:

- The ASP.NET process identity (as configured on the <**processModel**> element in web.config.

  This defaults to the local ASPNET account specified with the username "machine" and password "AutoGenerate".

- The authenticated caller's identity (if impersonation is enabled within web.config) as shown below.

```
<identity impersonate="true" />
```

  If you have not disabled anonymous access in IIS, this will be IUSR_MACHINE.

- A specified impersonation identity as shown below (although this is not recommended).

```
<identity impersonate="true" userName="Bob" password="password" />
```

More information

For more information about the identity used to run ASP.NET and the identity used to access local and network resources, see Chapter 8, ASP.NET Security.

Check the <authorization> Element

Confirm that the <**allow**> and <**deny**> elements are configured correctly.

- If you have <**deny users**="?" /> and you are using Forms authentication and/or IIS anonymous authentication, you must explicitly place an **IPrincipal** object into **HttpContext.User** or you will receive an access denied 401 response.
- Make sure the authenticated user is in the roles specified in <**allow**> and <**deny**> elements.

ASP.NET

Enable Tracing

ASP.NET provides quick and simple tracing to show the execution of events within a page and the values of common variables. This can be a very effective diagnostic aid. Use the page level **Trace** directive to turn on tracing, as shown below:

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs" AutoEventWireup="false"
Inherits="Test.WebForm1" Trace="true" %>
```

More information

For more information on the new tracing feature in ASP.NET see the Knowledge Base article Q306731, INFO: New Tracing Feature in ASP.NET.

Configuration Settings

Most application settings should be placed in web.config. The following list shows main security related settings that can be placed in web.config.

```
<authentication>

<authorization>

<trust>

<identity>
```

The following setting which controls the identity used to run the ASP.NET worker process (aspnet_wp.exe) must be located in machine.config.

```
<processModel>
```

Configuration settings for an application are always retrieved from the application's web.config file first. These override any equivalent settings within machine.config. If you want a particular setting to be applied to your application, explicitly configure the setting in the application's web.config file.

The main (and often only) web.config file for a particular application lives in its virtual directory root. Subdirectories can also contain web.config files. Settings in these files override the settings from web.config files in parent directories.

Determining Identity

Many security and access-denied problems relate to the identity used for resource access. The following code samples presented in this section can be used to help determine identity in Web pages, COM objects, and Web services.

For more information about .NET identity variables, see ASP.NET Identity Matrix in the Reference section of this guide.

Determining Identity in a Web Page

The following script can be used to gather security context-related information and indicates the identity being used to run a Web page.

To use this code, copy and paste it to create a file with a .aspx file extension. Copy the file to an IIS virtual directory and view the page from a browser.

```
<%@ Page language="c#" AutoEventWireup="true" %>

<%@ Import Namespace="System.Threading" %>

<%@ Import Namespace="System.Security.Principal" %>

<HTML>

  <HEAD>

    <title>WhoAmI</title>

  </HEAD>

  <body>

    <form id="WhoAmI" method="post" runat="server">

      <TABLE id=contextTable border=1>

        <TR>

          <TD align=middle colSpan=3 rowSpan="">

                HttpContext.Current.User.Identity</TD>

        </TR>

        <TR>

          <TD>Name</TD>

          <TD><asp:Label ID="contextName" Runat=server /></TD>

        </TR>

        <TR>

          <TD>IsAuthenticated</TD>

          <TD><asp:Label ID="contextIsAuth" Runat=server /></TD>

        </TR>

        <TR>

          <TD>AuthenticationType</TD>

          <TD><asp:Label ID="contextAuthType" Runat=server /></TD>

        </TR>
```

```
      </TABLE>

       <br><br>


      <TABLE id=windowsIdentityTable border=1>

        <TR>

          <TD align=middle colSpan=3
rowSpan="">WindowsIdentity.GetCurrent()</TD>

        </TR>

        <TR>

          <TD>Name</TD>

          <TD><asp:Label ID="windowsName" Runat=server /></TD>

        </TR>

        <TR>

          <TD>IsAuthenticated</TD>

          <TD><asp:Label ID="windowsIsAuth" Runat=server /></TD>

        </TR>

        <TR>

          <TD>AuthenticationType</TD>

          <TD><asp:Label ID="windowsAuthType" Runat=server /></TD>

        </TR>

      </TABLE>

      <br><br>


      <TABLE id=threadIdentityTable border=1>

        <TR>

          <TD align=middle colSpan=3

              rowSpan="">Thread.CurrentPrincipal.Identity</TD>

        </TR>
```

```
        <TR>
          <TD>Name</TD>
          <TD><asp:Label ID="threadName" Runat=server /></TD>
        </TR>
        <TR>
          <TD>IsAuthenticated</TD>
          <TD><asp:Label ID="threadIsAuthenticated" Runat=server /></TD>
        </TR>
        <TR>
          <TD>AuthenticationType</TD>
          <TD><asp:Label ID="threadAuthenticationType" Runat=server /></TD>
        </TR>
      </TABLE>
    </form>
  </body>
</HTML>
<script runat=server>
  void Page_Load(Object sender, EventArgs e)
  {
    IIdentity id = HttpContext.Current.User.Identity;
    if(null != id)
    {
      contextName.Text = id.Name;
      contextIsAuth.Text = id.IsAuthenticated.ToString();
      contextAuthType.Text = id.AuthenticationType;
    }
    id = Thread.CurrentPrincipal.Identity;
    if(null != id)
```

```
    {
      threadName.Text = id.Name;
      threadIsAuthenticated.Text = id.IsAuthenticated.ToString();
      threadAuthenticationType.Text = id.AuthenticationType;
    }
    id = WindowsIdentity.GetCurrent();
    windowsName.Text = id.Name;
    windowsIsAuth.Text = id.IsAuthenticated.ToString();
    windowsAuthType.Text = id.AuthenticationType;
  }
</script>
```

Determining Identity in a Web service

The following code can be used within a Web service to obtain identity information.

```
[WebMethod]
public string GetDotNetThreadIdentity()
{
  return Thread.CurrentPrincipal.Identity.Name;
}
[WebMethod]
public string GetWindowsThreadIdentity()
{
  return WindowsIdentity.GetCurrent().Name;
}
[WebMethod]
public string GetUserIdentity()
{
  return User.Identity.Name;
```

```
}

[WebMethod]

public string GetHttpContextUserIdentity()

{

  return HttpContext.Current.User.Identity.Name;

}
```

More information

- Security-related Knowledge Base articles
- Security-related Knowledge Base articles that deal with frequently seen error messages

Determining Identity in a Visual Basic 6 COM Object

The following method can be used to return the identity of a Visual Basic® 6 COM object. You can call Visual Basic 6.0 COM objects directly from ASP.NET applications through COM interop. The following method can be helpful when you need to troubleshoot access denied errors from your component when it attempts to access resources.

```
Private Declare Function GetUserName Lib "advapi32.dll" _

        Alias "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As
Long




Public Function WhoAmI()

   Dim sBuff    As String

   Dim lConst   As Long

   Dim lRet     As Long

   Dim sName    As String


   lConst = 199

   sBuff = Space$(200)

   lRet = GetUserName(sBuff, lConst)

   WhoAmI = Trim$(Left$(sBuff, lConst))

End Function
```

.NET Remoting

If a remote object is hosted in ASP.NET, and is configured for Windows authentication, you must specify the credentials to be used for authentication through the credentials property of the channel. If you do not explicitly set credentials, the remote object is called without any credentials. If Windows authentication is required, this will result in an HTTP status 401, access denied response.

To use the credentials associated with the current thread impersonation token (if the client thread is impersonating), or the process token (with no impersonation), use default credentials. This can be configured in the client-side configuration file using the following setting:

```
<channel ref="http" useDefaultCredentials="true" />
```

If an ASP.NET Web application calls a remote component and the Web application is configured for impersonation, the Web application must be using Kerberos or Basic authentication. All other authentication types can not be used in delegation scenarios.

If the Web application is not configured for impersonation, the process identity of the ASP.NET worker process is used. This is specified on the <**processModel**> element of machine.config and defaults to the local ASPNET account.

Note Ensure the process in running under an account that can be authenticated by the remote computer.

More Information

For more information about setting client-side credentials when calling remote components, see Chapter 11, .NET Remoting Security.

SSL

To troubleshoot SSL related problems:

- Confirm whether you can telnet to port 443 on the IP addresses of the client and server computer. If you cannot, this usually signifies that the sspifilt.dll is not loaded, or is the wrong version, or perhaps conflicts with other ISAPI extensions.
- Examine the certificate. If you can telnet to 443, check the certificates attribute using the browser's **View Certificate** dialog box. Check the certificate's effective and expiration dates, whether the common name is correct, and also what the Authority Information Access (AIA) or Certificate Revocation List (CRL) distribution point is.

  Confirm that you can browse directory to those AIA/CRL points successfully.

- If you are using a custom client application (and not a Web browser) to access an SSL-enabled Web site that requires client certificates, check that the client certificate is located in the correct store that the client application accesses.

  When you use a browser, the certificate must be in the interactive user's user store. Services or custom applications may load the client certificate from the machine store or a store associated with a service account's profile. Use the Services MMC snap-in (available when Certificate Services is installed), from the Administrative Tools program group to examine the contents of certificate stores.

More Information

See the following SSL related Knowledge Base articles.

- Q257591, Description of the Secure Sockets Layer (SSL) Handshake
- Q257586, Description of the Client Authentication Process During the SSL Handshake
- Q257587, Description of the Server Authentication Process During the SSL Handshake
- Q301429, HOWTO: Install Client Certificate on IIS Server for ServerXMLHTTP Request Object

- Q295070, SSL (https) Connection Slow with One Certificate but Faster with Others

IPSec

The following articles in the Knowledge Base provides steps for troubleshooting IPSec issues.

- Q259335, Basic L2TP/IPSec Troubleshooting in Windows

Auditing and Logging

Windows Security Logs

Consult the Windows event and security logs early on in the problem diagnostic process.

More information

For more information on how to enable auditing and monitoring events, see the Knowledge Base and article Q300958, HOW TO: Monitor for Unauthorized User Access in Windows 2000.

SQL Server Auditing

By default, logon auditing is disabled. You can configure this either through SQL Server™ Enterprise Manager or by changing the registry.

SQL Server log files are by default located in the following directory. They are text-based and can be read with any text editor such as Notepad.

```
C:\Program Files\Microsoft SQL Server\MSSQL\LOG
```

To enable logon auditing with Enterprise Manager

1. Start Enterprise Manager.
2. Select the required SQL Server in the left hand tree control, right-click and then click **Properties**.
3. Click the **Security** tab.
4. Select the relevant Audit level—**Failure**, **Success** or **All**.

To enable logon auditing using a registry setting

1. Create the following **AuditLevel** key within the registry and set its value to one of the REG_DWORD values specified below.

```
2.      HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\AuditLevel
```

3. Set the value of this key to one of the following numeric values, which allow you to capture the relevant level of detail.

   3—captures both success and failed login attempts.

   2—captures only failed login attempts.

   1—captures only success login attempts.

   0—captures no logins.

It is recommended that you turn on failed login auditing as this is a way to determine if someone is attempting a brute force attack into SQL Server. The performance impacts of logging failed audit attempts are minimal unless you are being attacked, in which case you need to know anyway.

You can also set audit levels by using script against the SQL Server DMO (Database Management Objects), as shown in the following code fragment.

```
Sub SetAuditLevel(Server As String, NewAuditLevel As SQLDMO_AUDIT_TYPE)
    Dim objServer As New SQLServer2
    objServer.LoginSecure = True  'Use integrated security
    objServer.Connect Server      'Connect to the target SQL Server
    'Set the audit level
    objServer.IntegratedSecurity.AuditLevel = NewAuditLevel
    Set objServer = Nothing
End Sub
```

From SQL Server Books online, the members of the enumerated type, SQLDMO_AUDIT_TYPE are:

```
SQLDMOAudit_All     3 Log all authentication attempts - success or failure
SQLDMOAudit_Failure 2 Log failed authentication
SQLDMOAudit_None    0 Do not log authentication attempts
SQLDMOAudit_Success 1 Log successful authentication
```

Sample log entries

The following list shows some sample log entries for successful and failed entries in the SQL Server logs.

Successful login using Integrated Windows authentication:

```
2002-07-06 22:54:32.42 logon     Login succeeded for user
'SOMEDOMAIN\Bob'. Connection: Trusted.
```

Successful login using SQL standard authentication:

```
2002-07-06 23:13:57.04 logon     Login succeeded for user
'SOMEDOMAIN\Bob'. Connection: Non-Trusted.
```

Failed login:

```
2002-07-06 23:21:15.35 logon     Login failed for user
'SOMEDOMAIN\BadGuy'.
```

IIS Logging

IIS logging can be set to different formats. If you use W3C Extended Logging, then you can take advantage of some additional information. For example, you can turn on Time Taken to log how long a page takes to be served. This can be helpful for isolating slow pages on your production Web site. You can also enable URI Query which will log Query String parameters, which can be helpful for troubleshooting GET operations against your Web pages. The figure below shows the Extended Properties dialog box for IIS logging.

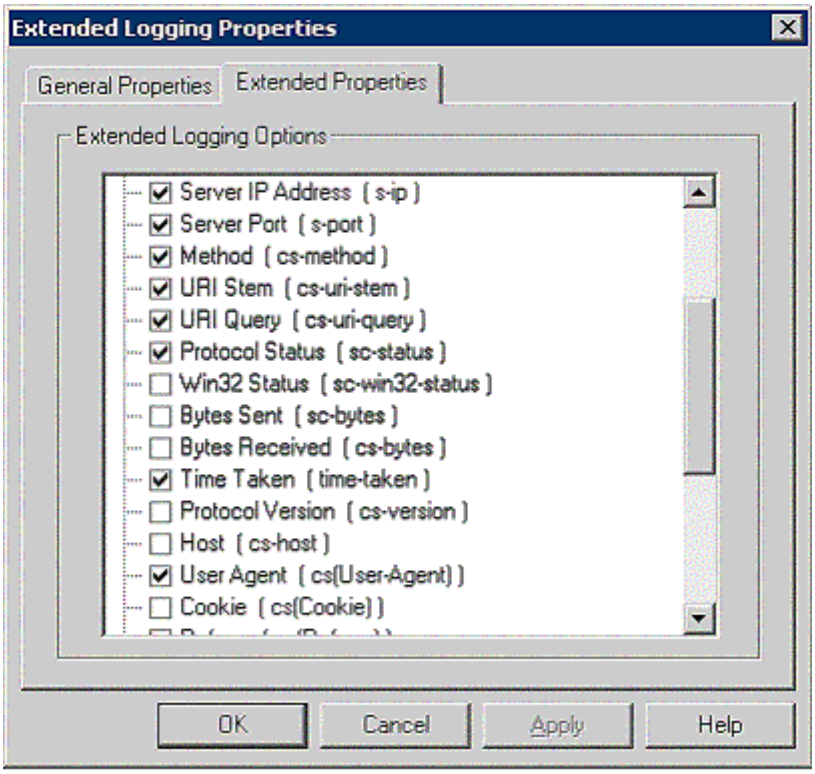

Figure 13.1. IIS extended logging properties

Troubleshooting Tools

The list of tools presented in this section can prove invaluable and will help you diagnose both security and non-security related problems.

File Monitor (FileMon.exe)

This tool allows you to monitor files and folders for access attempts. It is extremely useful to deal with file access permission issues. It is available from Sysinternals.com.

More information

For more information see the Knowledge Base article Q286198, HOWTO: Track 'Permission Denied' Errors on DLL Files.

Fusion Log Viewer (Fuslogvw.exe)

Fusion Log Viewer is provided with the .NET Framework SDK. It is a utility that can be used to track down problems with Fusion binding (see the .NET Framework documentation for more information).

To create Fusion logs for ASP.NET, you need to provide a log path in the registry and you need to enable the log failures option through the Fusion Log Viewer utility.

To provide a log path for your log files, use regedit.exe and add a directory location, such as e:\MyLogs, to the following registry key:

```
[HKLM\Software\Microsoft\Fusion\LogPath]
```

ISQL.exe

ISQL can be used to test SQL from a command prompt. This can be helpful when you want to efficiently test different logins for different users. You run ISQL by typing isql.exe at a command prompt on a computer with SQL Server installed.

Connecting by using SQL authentication

You can pass a user name by using the **-U** switch and you can optionally specify the password with the **-P** switch. If you don't specify a password, ISQL will prompt you for one. The following command, issued from a Windows command prompt, results in a password prompt. The advantage of this approach (rather than using the **-P** switch) is that the password doesn't appear on screen.

```
C:\ >isql -S YourServer -d pubs -U YourUser

Password:
```

Connecting by using Windows authentication

You can use the **-E** switch to use a trusted connection which uses the security context of the current interactively logged on user.

```
C:\ >isql -S YourServer -d pubs -E
```

Running a simple query

Once you are logged in, you can run a simple query, such as the one shown below.

```
1> use pubs

2> SELECT au_lname, au_fname FROM authors

3> go
```

To quit ISQL, type **quit** at the command prompt.

Windows Task Manager

Windows Task Manager on Windows XP and Windows Server 2003 allows you to display the identity being used to run a process.

To view the identity under which a process is running

1. Start **Task Manager**.
2. Click the **Processes** tab.
3. From the **View** menu, click **Select Columns**.
4. Select **User Name**, and click **OK**.

The user name (process identity) is now displayed.

Network Monitor (NetMon.exe)

NetMon is used to capture and monitor network traffic.

More information

See the following Knowledge Base articles:

- Q243270, HOW TO: Install Network Monitor in Windows 2000
- Q148942, HOW TO: Capture Network Traffic with Network Monitor
- Q252876, HOW TO: View HTTP Data Frames Using Network Monitor
- Q294818, Frequently Asked Questions About Network Monitor

There are a couple of additional tools to capture the network trace when the client and the server are on the same machine (this can't be done with Netmon):

- **tcptrace.exe**. Available from PocketSOAP.com. This is particularly useful for Web services since you can set it up to record and show traffic while your application runs. You can switch to Basic authentication and use tcptrace to see what credentials are being sent to the Web service.
- **packetmon.exe**. Available from AnalogX.com. This is a cut down version of Network Monitor, but much easier to configure.

Registry Monitor (regmon.exe)

This tool allows you to monitor registry access. It can be used to show read accesses and updates either from all processes or from a specified set of processes. This tool is very useful when you need to troubleshoot registry permission issues. It is available from Sysinternals.com.

WFetch.exe

This tool is useful for troubleshooting connectivity issues between IIS and Web clients. In this scenario, you may need to view data that is not displayed in the Web browser, such as the HTTP headers that are included in the request and response packets.

More information

For more information about this tool and the download, see the Knowledge Base article Q284285, How to Use Wfetch.exe to Troubleshoot HTTP Connections.

Visual Studio .NET Tools

The Microsoft .NET Framework SDK security tools can be found at the .NET Framework Tools Web site.

More information

See the following Knowledge Base articles:

- Q316365, INFO: ROADMAP for How to Use the .NET Performance Counters
- Q308626, INFO: Roadmap for Debugging in .NET Framework and Visual Studio

WebServiceStudio

This tool can be used as a generic client to test the functionality of your Web service. It captures and displays the SOAP response and request packets.

You can download the tool from the Web Service Tools page at GotDotNet.com.

Windows 2000 Resource Kits

Windows 2000 Resource Kits

Windows 2000 Resource Kit Free Tool Downloads