Microsoft

Web Service Security

Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0

patterns & practices

Contents

Forewards

Foreword by Alex Stamos and Scott Stender

As consultants for iSEC Partners, we have helped our customers develop and deploy Web service-based systems in environments that range from financial services to health care, and have helped multiple industry-leading independent software vendors integrate Web services into their products. We have shared our experience of testing and deploying secure Web services in multiple speaking opportunities, including in academic settings, at OWASP chapter and national meetings, and at conferences including SyScan and BlackHat.

In almost every presentation we have given, we are asked how to protect against security risks in a Web services world, and where developers can look for advice on writing secure Web services. Unfortunately, quality content and guidance has been hard to find.

The content provided by Microsoft's patterns & practices team addresses this dire need. The design and implementation guidance will help developers identify application-level security risks in their Web service deployments and implement standard practices to mitigate those risks. We recommend that Web service developers, particularly those using the .NET Framework, review this content and implement its suggestions to help improve security in an increasingly interconnected world. The design and implementation guidance provided in this guide increases understanding of this complex space, and should prove of significant use in the Web service development lifecycle.

Alex Stamos and Scott Stender
Founding Partners
iSEC Partners
November 2005

Alex Stamos is a founding partner of iSEC Partners, a strategic digital security organization. Alex is an experienced security engineer and consultant specializing in enterprise application security and has taught multiple classes in network and application security. Before he helped form iSEC Partners, Alex spent two years as a Managing Security Architect with @stake, performing advanced application security research and consulting. Alex has also run security for a large managed services company and has worked at a DoE national laboratory. He holds a BSEE from the University of California, Berkeley. Alex can be reached at alex@isecpartners.com.

Scott Stender is a founding partner of iSEC Partners, a strategic digital security organization. Scott brings with him several years of experience in large-scale software development and security consulting. Prior to helping form iSEC Partners, Scott specialized in application security consulting with @stake. In his research, Scott focuses on secure software engineering methodology and security analysis of core technologies. Most recently, Scott was published in the January-February 2005 issue of IEEE Security & Privacy, where he co-authored a paper entitled Software Penetration Testing and presented on Attacking Web Services at BlackHat USA 2005. He holds a BS in Computer Engineering from the University of Notre Dame. Scott can be reached at scott@isecpartners.com*.*

Foreword by Rudolph Araujo

Web services have, for a few years now, promised to be the future of the Internet and the World Wide Web. The ability to build rich federated environments that enable complex business-to-business scenarios and allow organizations to expose powerful line of business applications is tremendously exciting. All of this is possible using existing IT assets and investments and by adhering to universal standards that allow for interoperability between disparate technology solutions. With all that potential, the question often raised is why Web services have continued to remain on the "brink of deployment" in many organizations for so long.

In talking to many organizations, I have found that one of the biggest stumbling blocks tends to be the lack of a clear understanding of what it means to securely build and deploy a Web service, and create these truly federated scenarios. Customers complain about information overload with the host of three letter *ML acronyms and WS-* based standards. While a lot of these have been documented by various industry-wide bodies, little or no effort has been made in educating the practitioners in the thick of the battle — the architects, developers and testers building applications — about what they have to offer, how to use them and the tradeoffs and concerns to bear in mind while making design and implementation decisions.

Having been involved in this project right from the start as a technical reviewer, I believe that the Web Services Security guide from the patterns & practices group at Microsoft fills just this void. By providing accurate, timely, and relevant information, this guidance plays a crucial role in making some of the WS-Security standards easier to understand and thus allowing for an increase in the adoption and deployment of Web services. Further, by providing detailed but easy to comprehend explanations of the underlying protocols, such as Kerberos, the authors have ensured that even readers with a limited background in security will have adequate information and pointers, helping them gain valuable insights into this field.

Security personnel reading this guidance should focus on planning deployment scenarios based on the architectural and design patterns. The common scenario driven approach can prove to be of special value and relevance for this use case. On the other hand, developers are well advised to focus on the implementation patterns and technical supplements, which will introduce them to the topics and help them obtain a clear idea of the correct choices to make when faced with similar decisions in their own environments.

Rudolph Araujo
Principal Software Security Consultant
Foundstone Professional Services
November 2005

Rudolph Araujo is a Principal Software Security Consultant and trainer at Foundstone where he is responsible for creating and delivering the threat modeling and security code review service lines. He is also responsible for content creation and training delivery for Foundstone's Building Secure Software and Writing Secure Code — ASP.NET class. Rudolph has many years of software development experience on both UNIX and Windows environments in C/C++ and C#. Prior to Foundstone, Rudolph led the checks development team for BindView bv-Control for Internet Security — a vulnerability assessment product and was a software developer at Morgan Stanley. Rudolph's research interests also span the domain of Web service security and reliability. Rudolph holds a Masters Degree from Carnegie Mellon University with a focus on computer security and is the developer of Foundstone's .NET Security Toolkit, SSLDigger and Hacme Bank tools. Rudolph is also a Microsoft Visual Developer — Security MVP and a contributor to multiple journals such as Software Magazine where he writes a column on software security.

Foundstone

Foundstone Professional Services, a division of McAfee, offers a unique combination of services and education to help organizations continuously and measurably protect the most important assets from the most critical threats. Through a strategic approach to security, Foundstone helps organizations design and engineer secure software. Foundstone's services include source code audits, software design and architecture reviews, threat modeling and Web application penetration testing. For more information about Foundstone S3i services and training, go to www.foundstone.com/s3i.

Preface

Welcome to *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0*. The purpose of this guide is to help you successfully make the necessary decisions while you are securing your Web services. Like most decisions you make about solutions you are building, choosing the right security-related options are largely based on the requirements of the solution. The goal of this guide is to help you quickly make the most appropriate security decisions in the context of your solution's requirements while providing the rationale and education for each option. There are three different ways you can navigate this guide.

- **Read it from start to finish**. You will learn the most by doing this, but some of what you learn may not apply to the solutions you are building.
- **Use the decision matrices**. These exist in the chapter introductions and will help you refine your options for meeting your security needs.
- **Find a similar scenario**. If your solution resembles one of the four scenarios described in the Introduction, you can start by reading the sections that apply to that scenario.

Intended Audience

The target audiences for this guide are architects and developers who are designing or implementing Web services. Specifically, it is assumed that you either have experience designing and developing Web service solutions or solutions with security requirements (such as authentication, authorization, or encryption). This assumption does not mean this content may not be useful to individuals who do not meet these assumptions — it will just take them longer to realize the benefits of this guidance.

To experience the most value from this guide, you should have a basic understanding of how entities in a distributed application interact, and the security considerations of those interactions. This guidance does not explicitly cater to security experts, but an understanding of various security topics such as authentication, authorization, encryption, and digital signatures will help your comprehension. Likewise, a deep understanding of Web services is not required, but it can help.

How This Guide Is Organized

The majority of the content in this guide is presented in the form of patterns. A pattern describes a recurring problem that occurs in a particular situation and — based on a set of guiding forces called requirements — recommends a solution. Patterns exist at different levels of abstraction, including:

- **Architecture patterns**. These describe how to structure an application at the highest level.
- **Design patterns**. These describe how to structure subsystems or components within a system.
- **Implementation patterns**. These describe low-level patterns that are specific to a particular platform. In this case, using the Microsoft .NET Framework and Web Services Enhancements (WSE) 3.0.

A pattern is usually described with the following key elements:

- **Name**. The name is the simplest label that you can identify that captures what you are trying to achieve in solving the problem.
- **Context**. The context sets the stage of what artifacts within a problem domain you are working within.
- **Problem**. The problem is what you want to achieve or deal with within the context.
- **Forces**. The forces are identified key elements that must be handled in the context and affect the problem. Forces are conditions that exist within the context.
- **Solution**. The solution is a way to resolve the forces to solve the problem within a context. When you start looking at how you must deal with those forces, you end up with a resulting context.

The patterns in this guide are grouped into two main parts: Part I: Core Web Service Security Patterns and Part II: Additional Web Service Security Patterns and Guidance. Part I contains a core set of patterns that are often inter-related and used together. The implementation of these patterns is demonstrated by composite implementation patterns, which are patterns implemented in combination with other related patterns. Part II contains additional patterns, which should also be applied in many cases — but they can typically be applied after you have selected the core patterns you are implementing.

Note: This guide is intended as an additional resource to the Microsoft patterns & practices *Improving Web Application Security: Threats and Countermeasures* guide, which incorporates detailed information about how to determine your security requirements using a technique called threat modeling.

Community

This guide, like many patterns & practices deliverables, is associated with a community workspace. On this community workspace, you can post questions, provide feedback, or connect with other users for sharing ideas. Community members can also download additional content such as extensions, QuickStarts, and training material and can provide feedback that will help Microsoft plan and test future patterns.

Access to the Web Service Security community is available from the following Web site: *http://go.microsoft.com/fwlink/?LinkId=57044*.

Feedback and Support

Questions? Comments? Suggestions? To provide feedback about this guidance, or to get help with any problems, visit the Web Service Security community workspace. The message board on the community site is the preferred feedback and support channel because it allows you to share your ideas, questions, and solutions with the entire community. Alternatively, you can send e-mail directly to the Microsoft patterns & practices team at *devfdbck@microsoft.com*, although we are unable to respond to every message.

The Team Who Brought You This Guide

Thanks to the following individuals who assisted in the content development, QuickStart development, test, and documentation experience:

- **Lead authors**: Jason Hogg, Don Smith, Fred Chong, Microsoft Corporation; Dwayne Taylor, Lonnie Wall, RDA Corporation; and Paul Slater, Wadeware LLC.
- **Contributing authors**: Tom Hollander, Wojtek Kozaczynski, Microsoft Corporation.
- **Test team**: Larry Brader, Microsoft Corporation; Sajjad Nasir Imran, Mohanakrishan Shankar, Dhamotharan Bethanasamy, Subha Vaitheeswaran, Muralidharan C Narayanan, Venkat Narayan S., Sumit Baurai, Infosys Technologies Ltd.
- **Development team**: Diego Gonzalez, Pablo Cibraro, Ariel Szklarkiewicz, Lagash Systems SA.
- **Pattern workshop facilitator**: Ward Cunningham.
- **Editors and graphic artist**: Nelly Delgado, Microsoft Corporation; Sharon Smith, Linda Werner & Associates; Tina Burden McGrayne, Melissa Seymour, TinaTech Inc.; John Cobb, Wadeware LLC; Claudette Siroky, CI Design Studio.

Many thanks to the following individuals who provided invaluable feedback and assistance:

- **External reviewers**: Rudolph Araujo, Foundstone, Inc.; Scott Stender, Alex Stamos, Information Security Partners, LLC; Dr. Ivan Djordjevic, Security Research Centre, BT GCTO; David Mowers, Securitay, Inc.; Steve Stefanovich, RDA; Edward Bakker, LogicaCMG; Norman Headlam, Fidelity Investments; Jörg Bartholdt, Siemens AG; Anil John, Johns Hopkins University—Applied Physics Lab; Julia Lerman, The Data Farm; Keith Brown, Pluralsight; Vivek Vaid, Thoughtworks; Jason Ward; Munawar Hafiz, Patrick Delfert, University of Illinois at Urbana-Champaign; David Francis, HBOS Plc.
- **Microsoft reviewers and contributors**: Chris Keyser, Marc Goodner, David Trowbridge, Jonathan Wanagel, Mark Fussell, Jan Alexander, Sidd Shenoy, HongMei Ge, Tomasz Janczuk, David M. Bradley, Vijay Gajjala, Paul Leach, Hervey Wilson, Doug Walter, Martin Gudgin, Kannan C. Iyer, Heikki Ritvanen, J.D. Meier, Ron Jacobs, Doug Orange, Vittorio Bertocci, Kirk Allen Evans, Sam Keall, Vajira Weerasekera, Matt Deacon, Andy Gordon, Karthik Bhargavan, Cedric Fournet.
- Finally, thanks to Alan Ridlehoover, Blaine Wastell, Brad Wilson, Carlos Farre, Darrell Snow, Edward Jezierski, Edward Lafferty, Eugenio Pace, J.D. Meier, Ken Perilman, Michael Kropp, Mohammad Al-Sabt, Naveen Yajaman, Per Vonge Nielsen, Peter Provost, Rick Maguire, RoAnn Corbisier, Sanjeev Garg, Scott Densmore, Shaun Hayes, Srinath Vasireddy, Steve Elston, Vicky Titus, William Loeffler, Microsoft Corporation.

For more information about Web service security, see the following patterns & practices team blogs:

- Jason Hogg: *http://blogs.msdn.com/thehoggblog*
- Don Smith: *http://blogs.msdn.com/donsmith*

Introduction

To design, develop, and deploy secure Web services, architects and developers must learn new technologies and consider new threats associated with exposing functionality on potentially unsecured networks. To prepare you to meet these challenges today (both now on Microsoft® Web Services Enhancements 3.0 and in the future with Windows® Communication Foundation [WCF]), the Microsoft patterns & practices team has created *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0*.

Architects and developers responsible for Web service security have a considerable number of options available to them. These options are further complicated by the fact that different projects and different organizations have different security requirements.

To help you consider alternative approaches to securing your Web services, this guide provides a scenario-driven approach to demonstrate situations where different security patterns are successful. The guide also combines a series of decision matrices to assist you in applying your own criteria to use the Web service security patterns to meet the requirements of your environment.

Overview

This guidance supports the following major phases of a software development life cycle:

- **Evaluation**. This phase includes four common Web service security scenarios, an analysis of their key requirements, and a summary of how the decision matrices were used to select a series of design and architecture patterns to meet each scenario's requirements.
- **Design**. This phase includes the security decision matrices that you can combine with architectural and design patterns to assist you in making key design choices.
- **Implementation**. This phase includes composite patterns and implementation patterns that provide in-depth implementation details, including code examples that you can customize to meet the needs of your environment. Implementation patterns are provided for isolated challenges, and where appropriate, composite patterns organize critical patterns together.
- **Deployment**. This phase includes composite patterns and technical supplements that provide additional information to help you understand critical challenges in the deployment life cycle.

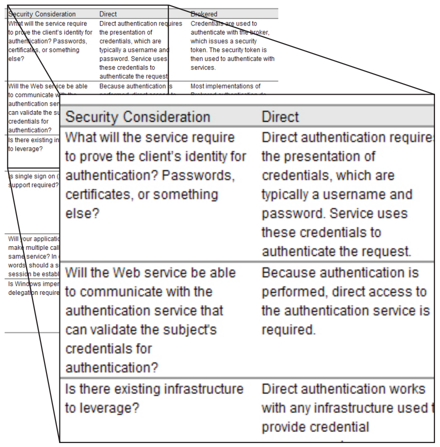
Key guidance topics that the *Web Service Security* guide discusses include:

- Choosing between message layer security and transport layer security.
- Choosing a client authentication technology, from basic direct authentication to more sophisticated brokered solutions, including an in-depth look at X.509 certificates, using the Kerberos version 5 protocol, and solutions involving a Security Token Service (STS).
- Protecting confidentiality of messages.
- Detecting tampered messages.
- Prventing the processing of replayed messages.
- Accessing remote resources and flowing identities across tiers.
- Preventing exceptions from revealing sensitive implementation details.
- Protecting Web services from malformed or malicious messages.

This guidance is designed to assist those who currently use the Microsoft Visual Studio® 2005 development system and WSE 3.0. It will also be of significant use to architects and developers who plan to provide solutions using WCF.

Navigating the Web Service Security Guide

There are many different options available to help secure your Web services, and different organizations have different criteria that drive their security decisions. This guide can of course be read from start to finish, and while this is the most comprehensive approach, you may find that you only require guidance in specific areas. In an attempt to make the guidance more productive for you, this guide uses decision matrices to help highlight key criteria that should be considered when selecting one approach over another. Figure 1 illustrates an excerpt from the authentication decision matrix that helps lead you through the process of choosing between the direct authentication and brokered authentication techniques.



Security Consideration	Direct
What will the service require to prove the client's identity for authentication? Passwords, certificates, or something else?	Direct authentication require the presentation of credentials, which are typically a username and password. Service uses these credentials to authenticate the request.
Will the Web service be able to communicate with the authentication service that can validate the subject's credentials for authentication?	Because authentication is performed, direct access to the authentication service is required.
Is there existing infrastructure to leverage?	Direct authentication works with any infrastructure used t provide credential

Figure 1
An excerpt from the authentication decision matrix

Decision matrices are included in the introductions to many of the chapters in this guide. They can help you select which of the following to use:

- Direct authentication or brokered authentication.
- The Kerberos protocol, X.509 certificates in a Public Key Infrastructure (PKI), or a Security Token Service (STS).
- Message layer security or transport layer security.
- Message protection requirements.
- Resource access techniques.

This guide also contains other tools to help you get the most out of the guidance. The Appendix to this guide includes a "Problem/Solution Index," where you can map specific problems to sections in the guide. This introduction also includes scenarios that share commonalities with the majority of Web services that companies are building today. The "Common Scenarios" section introduces four different scenarios that provide examples of common Web service interactions.

If you choose to read the chapters in this guide sequentially, you will benefit by understanding the importance of the sequence. The guide is divided into two parts. Part I covers core Web service security patterns, and includes three chapters:

- Chapter 1, "Authentication Patterns," helps you make the most appropriate decision regarding authentication, because many of the following Web service decisions depend on your choices for authentication.
- Chapter 2, "Message Protection Patterns," helps you understand the different message protection capabilities to determine which ones are appropriate to meet your requirements.
- Chapter 3, "Implementing Transport and Message Layer Security," helps you decide between using message layer security and transport layer security, and provides you with the implementation patterns in WSE 3.0.

Part II of the guide covers additional Web service security patterns and guidance. This information should normally be considered after you have already reviewed Part I. Part II consists of four chapters, which discuss resource access patterns, service boundary protection patterns, service deployment patterns, and technical supplements.

Additional useful information appears in the Appendix. The Appendix contains information about Web service interoperability, the Policy Analyzer for WSE 3.0, and a white paper about using patterns as a common vocabulary for individuals involved in the Information Technology industry. A glossary of commonly used terms and the "Problem/Solution Index" are also included in the Appendix.

Important Concepts

There are some important concepts you should understand before reviewing the different scenarios. These include:

- **Brokered authentication**. This is a type of authentication where a trusted authority is used to broker authentication services between a client and a service. You can use a broker to perform authentication.
- **Client**. The client accesses the Web service. The client provides credentials for authentication during the request to the Web service.
- **Credentials**. A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential.
- **Direct authentication**. A type of authentication where the service validates credentials directly with an identity store, such as a database or directory service.
- **Impersonation**. The act of assuming a different identity on a temporary basis so that a different security context or set of credentials can be used to access a resource.
- **Message layer security**. Represents an approach where all the information that is related to security is encapsulated in the message. In other words, with message layer security, the credentials are passed in the message.
- **Mutual authentication**. This is a form of authentication where the client authenticates the server in addition to the server that authenticates the client.
- **Security token**. A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential. Most security tokens will also contain additional information that is specific to the authentication broker that issued the token.
- **Service**. A Web service that requires authentication.
- **Transport layer security**. Represents an approach where security protection is enforced by lower level network communication protocols.
- **Trusted subsystem**. This is a process where a trusted business identity is used to access a resource on behalf of the client. The identity could belong to a service account or it could be the identity of an application account created specifically for access to remote resources.

For a complete list of security terms used throughout this guidance, see the Glossary in the Appendix.

Common Scenarios

To familiarize you with the guidance provided by the decision matrices, this section introduces several common scenarios that illustrate Web service solutions. It provides some requirements for each scenario, and then extracts the key points from the relevant decision matrices that were used for deciding one approach over another. This approach should help you better understand how to use the decision matrices to determine appropriate solutions for your own requirements. The scenarios also aim to show how a series of patterns can be used together to increase the security of your Web applications.

The following four scenarios provide examples of common Web service interactions:

- **Public Web service**. This scenario describes the decision criteria used to choose transport layer confidentiality with HTTPS and **UsernameToken** support in WSE 3.0 for authentication.
- **Intranet Web service**. This scenario describes the decision criteria used to choose message layer security with the Kerberos protocol for an internal banking solution. It also provides a high-level description of the Kerberos design.
- **Internet business-to-business**. This scenario describes a business-to-business solution that uses message layer security with the Kerberos protocol within the organization and X.509 certificates between businesses.
- **Multiple Internet Web services**. This scenario describes the decision criteria used to choose a Security Token Service (STS) for a travel agency application that is accessible from the Internet. This section also describes how both direct authentication and brokered authentication are used to implement the solution.

Note: The scenarios are just examples to illustrate different security considerations as you navigate through the *Web Service Security* guide. They are not meant to represent the only way to implement these types of Web service solutions. Instead, they show you how information related to authentication, message protection, and protection scope can be used to navigate through the various patterns.

Each scenario starts with a high-level description of the application followed by a Web service profile that identifies the business requirements for the application. Some of these requirements are also included as security considerations in the solution approach.

Following the high-level description is a solution approach that examines factors related to the existing security infrastructure, organization security policies, and security threats that can lead to other security considerations. Each of these security considerations is categorized into three areas that directly relate to the chapter content in the Core Web Service Security Patterns part of this guidance.

The three categories and the chapters that they relate to are:

- **Authentication**. This category is associated with Chapter 1, "Authentication Patterns."
- **Message Protection**. This category is associated with Chapter 2, "Message Protection Patterns."
- **Protection Scope**. This category is associated with Chapter 3, "Implementing Transport and Message Layer Security."

The security consideration related to each category is then used to navigate through the appropriate decision matrices in the introduction to each chapter.

The last section in each scenario identifies patterns that were used for the candidate solution and how the solution was implemented. Additional patterns that could have been considered as part of the solution are also identified.

Public Web Service Scenario

A large clothing distributor uses Web services to provide catalog information to merchants that provide online shopping services. The merchants access the Web service from their Web applications to display current items available from the distributor.

Figure 2 illustrates how the online merchants access the Web service.

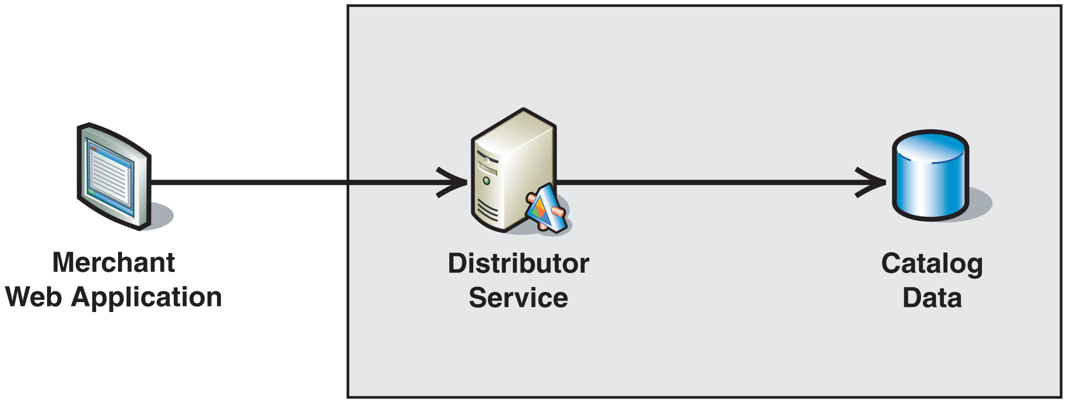


Figure 2
A distributor Web service

The following sections provide an overview of the distributor Web service requirements.

Distributor Web Service Profile

A distributor Web service has the following requirements:

- The merchant Web application requires direct access to the distributor's Web service.
- Merchants accessing the Web service must be authenticated.
- Data passed between the merchant and distributor contains some information, such as merchant account information, that must be protected.

Solution Approach

Table 1 lists security factors that were considered for the distributor Web service and how each factor maps to a specific category.

Table 1: Distributor Web Service Factors

Factor	Security consideration	Category
Security infrastructure	Merchant accounts are stored in a custom database or directory service.	Authentication
Security threats	Message data is sensitive and must be protected against unauthorized access.	Message Protection

The information in Table 1 is combined with business requirements related to security, and then it is grouped by category. Each category represents one or more decision matrices. The next step is to apply the security considerations to the appropriate matrices to make security decisions. In this example, you would examine the authentication decision matrices in Chapter 1, "Authentication Patterns," and the message protection decision matrix in Chapter 2, "Message Protection Patterns."

Table 2 provides a summary of the decisions that were made after applying the security considerations from Table 1, and the related business requirements to the appropriate decision matrices.

Table 2: Summary of Security Decisions

Factor	Security consideration	Security decision
Authentication	Merchant accounts are stored in a custom database or directory service.	**UsernameToken** can be used with custom authentication, Windows authentication or any other directory service that provides authentication.
Authentication	Merchants accessing the Web service must be authenticated.	**UsernameToken** provides the ability to authenticate the merchants.
Message Protection	Message data is sensitive and must be protected against unauthorized access.	HTTPS protects the message data while in transit between the merchant and distributor.

Candidate Solution

This solution uses the following patterns to implement direct authentication with **UsernameToken** and HTTPS to provide message protection:

- Direct Authentication in Chapter 1, "Authentication Patterns."
- Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
- Trusted Subsystem in Chapter 4, "Resource Access Patterns."

Figure 3 illustrates the distributor Web service security solution.

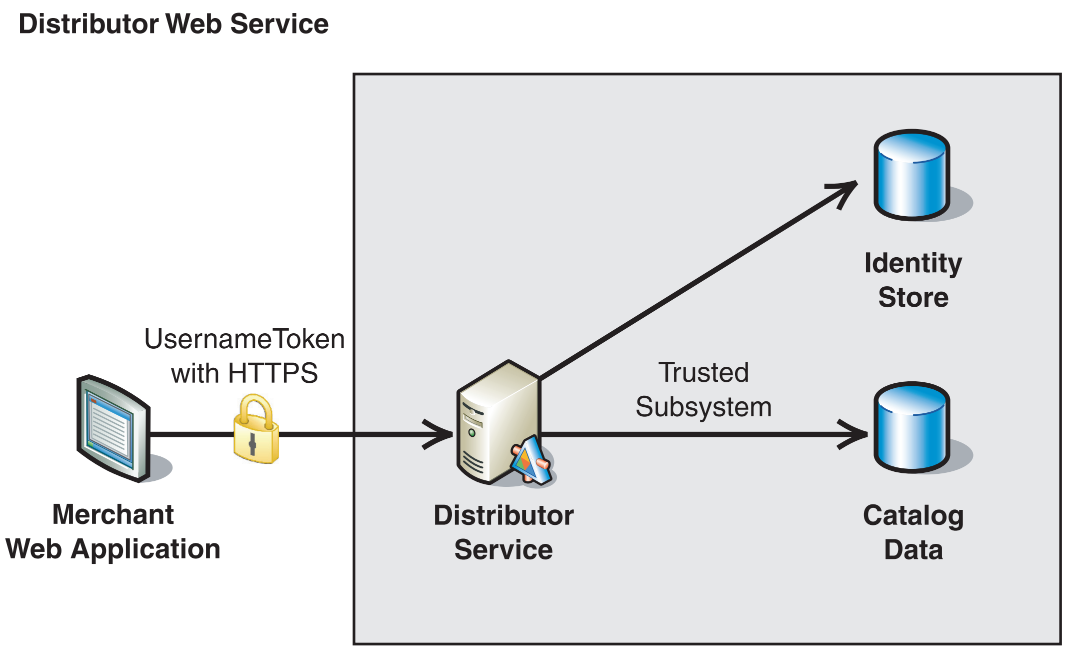


Figure 3
The security solution for a distributor Web service

The distributor Web service security solution is implemented in the following way:

- The distributor Web service uses a server certificate to establish secure communications with the merchant Web application using HTTPS.
- The merchant Web application passes a **UsernameToken** to the distributor Web service for authentication.
- The **UsernameToken** information is used to authenticate the merchant Web application.
- The distributor Web service uses a trusted subsystem to access catalog data.

Additional patterns that could have been considered include:

- Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
- Message Validator in Chapter 5, "Service Boundary Protection Patterns."
- Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."

Intranet Web Service Scenario

A national bank uses Web services to provide operations that are accessed by an internal banking application. Figure 4 illustrates how the banking application accesses the Web service.

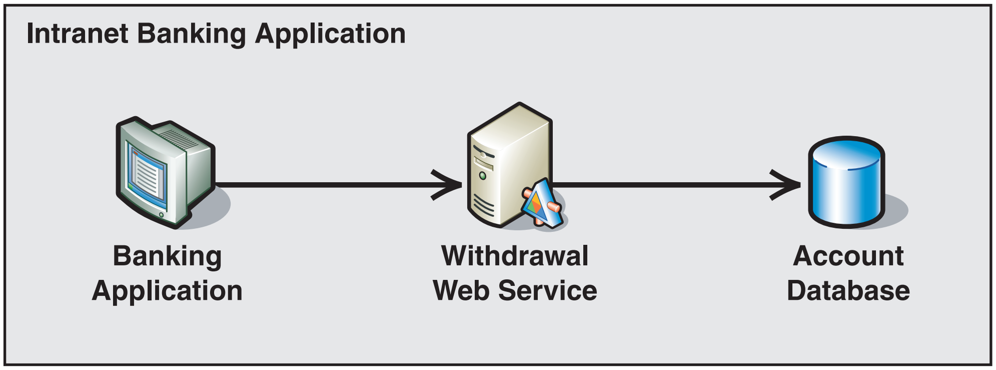


Figure 4
An intranet banking application Web service

The banking application is a Windows client that directly accesses a Web service. The Web services access a bank account database for information. The following sections provide an overview of the banking application requirements.

Banking Application Profile

The banking application has the following features:

- The banking application is used in bank branches.
- The user of the application is a customer service representative (CSR).
- The CSR must be authenticated as a valid user to use the banking application.
- Banking regulations require that the account activities that the CSR performs must be audited.

Solution Approach

Table 3 lists factors that were considered for the banking intranet scenario and how each factor maps to a specific category.

Table 3: Intranet Banking Application Factors

Factor	Security consideration	Category
Security infrastructure	Active Directory® directory service is implemented on a computer running Microsoft Windows Server™ 2003.	Authentication
Security infrastructure	CSR users are located in Active Directory.	Authentication
Organization security policies	Mutual authentication is required for all Web service interactions.	Authentication
Organization security policies	Applications must support single sign on (SSO) capabilities.	Authentication
Security treats	Message data is sensitive and must be protected against unauthorized access.	Message Protection
Security treats	The message must not be tampered with during transit.	Protection Scope

The information in Table 3 is combined with business requirements related to security, and then it is grouped by category. Each category represents one or more decision matrices. The next step is to apply the security considerations to the appropriate matrices to make security decisions. In this example, you would examine the authentication decision matrices in Chapter 1, "Authentication Patterns," the message protection decision matrix in Chapter 2, "Message Protection Patterns," and the protection scope decision matrix in Chapter 3, "Implementing Transport and Message Layer Security."

Table 4 provides a summary of the decisions that were made after applying the security considerations from Table 3 and the related business requirements to the appropriate decision matrices.

Table 4: Summary of Security Decisions

Category	Security consideration	Security decision
Authentication	CSR users are located in Active Directory on a computer running the Windows Server 2003 operating system.	Active Directory supports the use of the Kerberos protocol.
Authentication	Applications must support SSO capabilities.	The Kerberos protocol provides support for SSO capabilities.
Authentication	Mutual authentication is required.	Because the **KerberosToken** contains both requestor and service information, it can be used for mutual authentication.
Authentication	Account activities carried out by CSR users must be audited.	The Kerberos protocol also supports impersonation and delegation, which means that auditing can be performed.
Message Protection	Message data is sensitive and must be protected against unauthorized access.	The **KerberosToken** can be used to encrypt a message.
Protection Scope	The message must not be tampered with during transit.	The **KerberosToken** can be used to sign a message, which provides data integrity and data origin authentication.

Candidate Solution

This solution uses the following patterns to implement message layer security with the Kerberos protocol:

- Brokered Authentication in Chapter 1, "Authentication Patterns."
- Brokered Authentication: Kerberos in Chapter 1, "Authentication Patterns."
- Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Data Origin Authentication in Chapter 2, "Message Protection Patterns."
- Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Figure 5 illustrates the intranet banking application security solution.

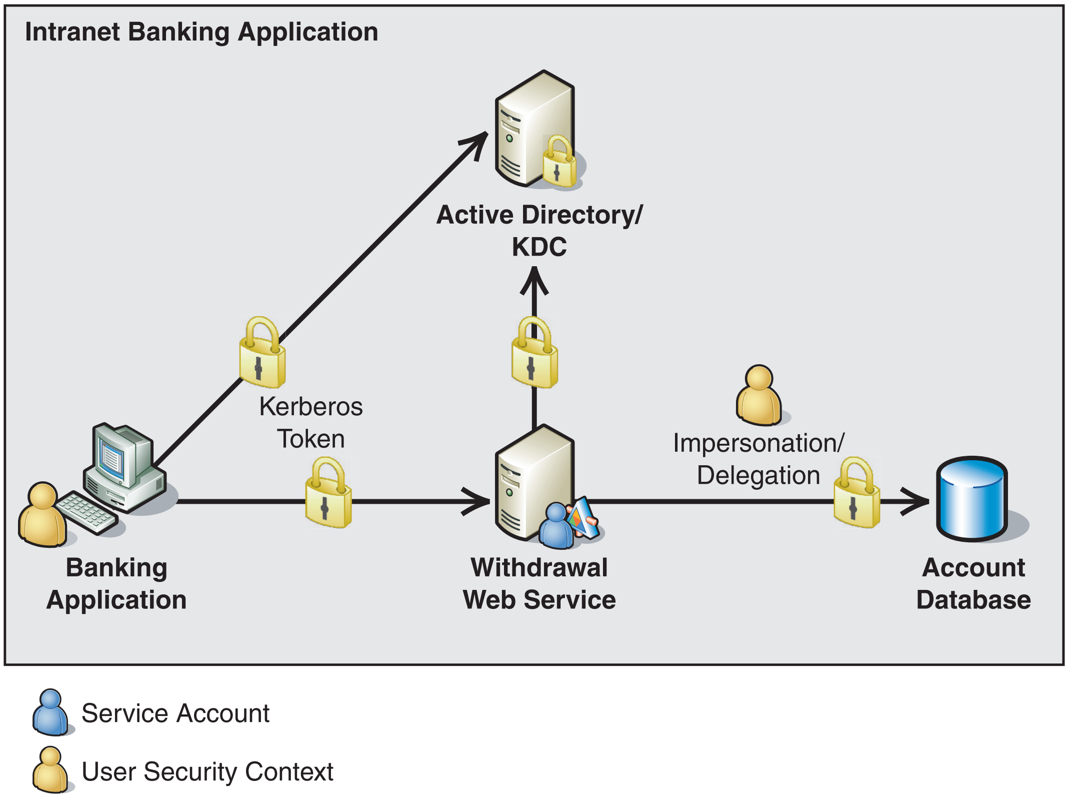


Figure 5
The security solution for the intranet banking application Web service

The intranet banking security solution is implemented in the following way:

- The user's credentials are used to obtain a security token from the Kerberos Key Distribution Center (KDC) implemented in Active Directory.
- The security token is used to sign and encrypt messages sent to the service.
- The security token is used to obtain additional information about the user from Active Directory.
- Impersonation with delegation is used to access the database.

Note: For information about impersonation and constrained delegation, see Chapter 4, Resource Access Patterns.

Additional patterns that could have been considered include:

- Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."
- Message Validator in Chapter 5, "Service Boundary Protection Patterns."

Internet Business-to-Business Scenario

A supply chain application uses internal Web services to perform operations. The internal Web services may need to access external Web services provided by another company. Figure 6 provides a high-level view of a procurement operation.

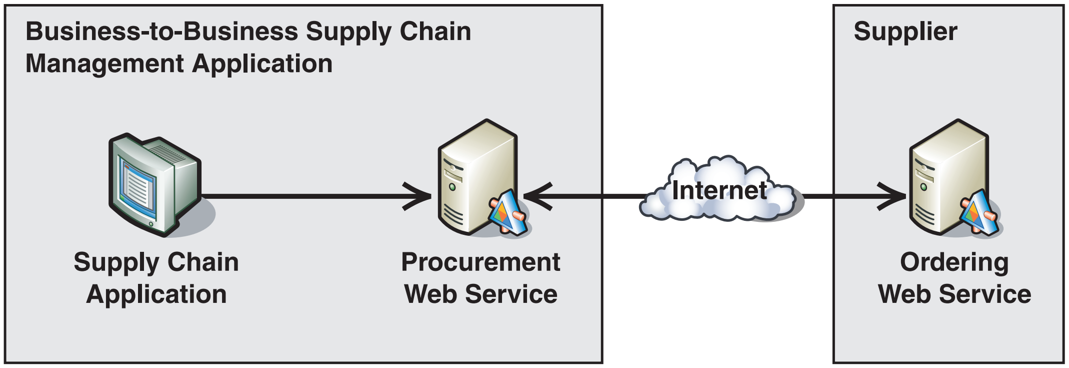


Figure 6
A business-to-business supply chain management application

Figure 6 illustrates an operation where the supply chain application interacts with the procurement Web service through an intranet. The procurement Web service accesses an external ordering Web service over the Internet. The following sections provide an overview of the supply chain application requirements.

Supply Chain Management Application Profile

The supply chain management application has the following features:

- The manufacturing company gets parts from a business partner.
- Parts are ordered through an internal line-of-business supply chain management application.
- Factory floor supervisors are the users of the application.
- The application communicates with a procurement Web service that places orders with an ordering Web service hosted by the supplier. This way, only the two Web services have to agree on the external service contract.
- The procurement Web service is one of a few other internal Web services that the supply chain management application uses. Maintaining an SSO user experience is an important requirement.

Solution Approach

There are actually two parts in this scenario to analyze: the intranet communication between the supply chain application and the procurement Web service, and the Internet communication between the procurement Web service and the ordering Web service.

Table 5 provides a list of factors considered for the Internet business-to-business scenario and how each factor maps to a specific category.

Table 5: Internet Business-to-Business Application Factors

Factor	Security consideration	Category
Security infrastructure	Active Directory is implemented on a computer running Windows Server 2003.	Authentication
Security infrastructure	Application users are located in Active Directory.	Authentication
Security infrastructure	The external Web service is hosted in an unknown environment.	Authentication
Organization security policies	Mutual authentication is required for all Web service interactions.	Authentication
Organization security policies	Applications must support SSO capabilities.	Authentication
Security treats	Factory parts and associated pricing information is sensitive. As a result, the data must be protected against unauthorized access.	Message Protection
Security treats	The message must not be tampered with during transit.	Protection Scope

The information in Table 5 is combined with the business requirements related to security, and then it is grouped by category. Each category represents one or more decision matrices. The next step is to apply the security considerations to the appropriate matrices to make security decisions. In this example you would examine the authentication decision matrices in Chapter 1, "Authentication Patterns," the message protection decision matrix in Chapter 2, "Message Protection Patterns," and the protection scope decision matrix in Chapter 3, "Implementing Transport and Message Layer Security."

Table 6 provides a summary of the decisions that were made after applying the security considerations from Table 5 and the related business requirements to the appropriate decision matrices.

Table 6: Summary of Security Decisions

Category	Security consideration	Security decision
Authentication	Supply chain application users are located in Active Directory on a computer running Windows Server 2003.	Within the intranet, the Kerberos protocol is supported by Active Directory.
Authentication	Applications must support SSO capabilities.	The Kerberos protocol provides support for SSO capabilities within the supply chain application intranet.
Authentication	The external Web service is hosted in an unknown environment.	Interaction between the internal and external Web services does not require the credentials of the user. As a result, an alternate form of authentication can be used, such as message layer security with X.509 certificates.
Authentication	The external Web service is hosted in an unknown environment.	X.509 certificates represent a well-known protocol that supports interoperability with other platforms.
Authentication	Mutual authentication is required.	Both message layer security with the Kerberos protocol and message layer security with X.509 certificates support mutual authentication.
Message Protection	Data must be protected against unauthorized access.	Message layer security with the Kerberos protocol supports both data confidentiality and data origin authentication. Message layer security with X.509 certificates also supports data confidentiality and data origin authentication.
Protection Scope	The message must not be tampered with during transit.	The **KerberosToken** can be used to sign a message, which provides data integrity and data origin authentication.

Candidate Solution

This solution uses the following patterns to implement message layer security with the Kerberos protocol in the intranet environment, and message layer security with X.509 certificates between the procurement Web service and the ordering Web service:

- Brokered Authentication in Chapter 1, "Authentication Patterns."
- Brokered Authentication: Kerberos in Chapter 1, "Authentication Patterns."
- Brokered Authentication: X.509 PKI in Chapter 1, "Authentication Patterns."
- Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Data Origin Authentication in Chapter 2, "Message Protection Patterns."
- Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
- Implementing Message Layer Security with X.509 Certificates in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
- Perimeter Service Router in Chapter 6, "Service Deployment Patterns."

Figure 7 illustrates the security solution that was developed for the supply chain management application.

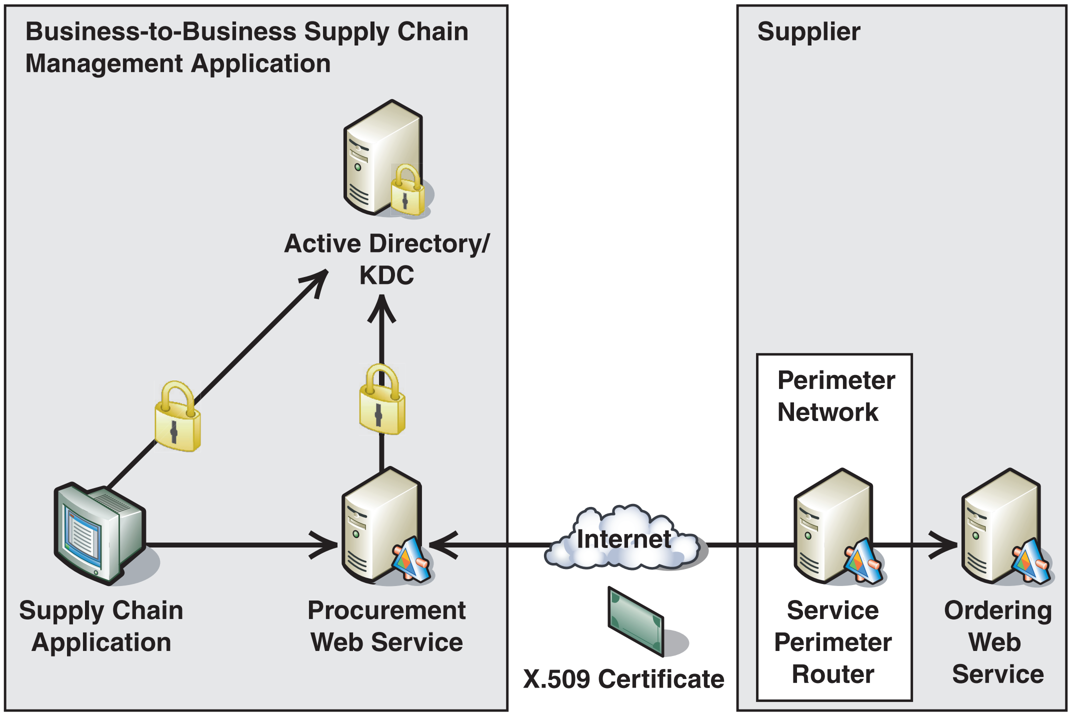


Figure 7
The security solution for the supply chain management application

The supply chain management security solution is implemented in the following way:

- The user's credentials are used to obtain a security token from the Kerberos KDC implemented in Active Directory.
- The security token is used to sign and encrypt messages sent to the service.

The supplier's security solution is implemented in the following way:

- X.509 certificates are issued and imported into appropriate certificate stores.
- X.509 certificates are used to provide mutual authentication, data confidentiality, and data origin authentication for interactions between the procurement Web service and the ordering Web service.
- A perimeter service router is used to accept requests from the supply chain application and send them to the ordering Web service.

Note: For information about configuring X.509 certificates, see the X.509 Technical Supplement in Chapter 7, "Technical Supplements."

Additional patterns which could have been considered include:

- Message Validator in Chapter 5, "Service Boundary Protection Patterns."
- Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."

Multiple Internet Web Services Scenario

A travel booking franchise provides a Web application that travel agents can use to search for and book travel packages. The Web application uses several Web services to perform the operations of searching for and booking packages. Figure 8 illustrates a high-level view of the configuration.

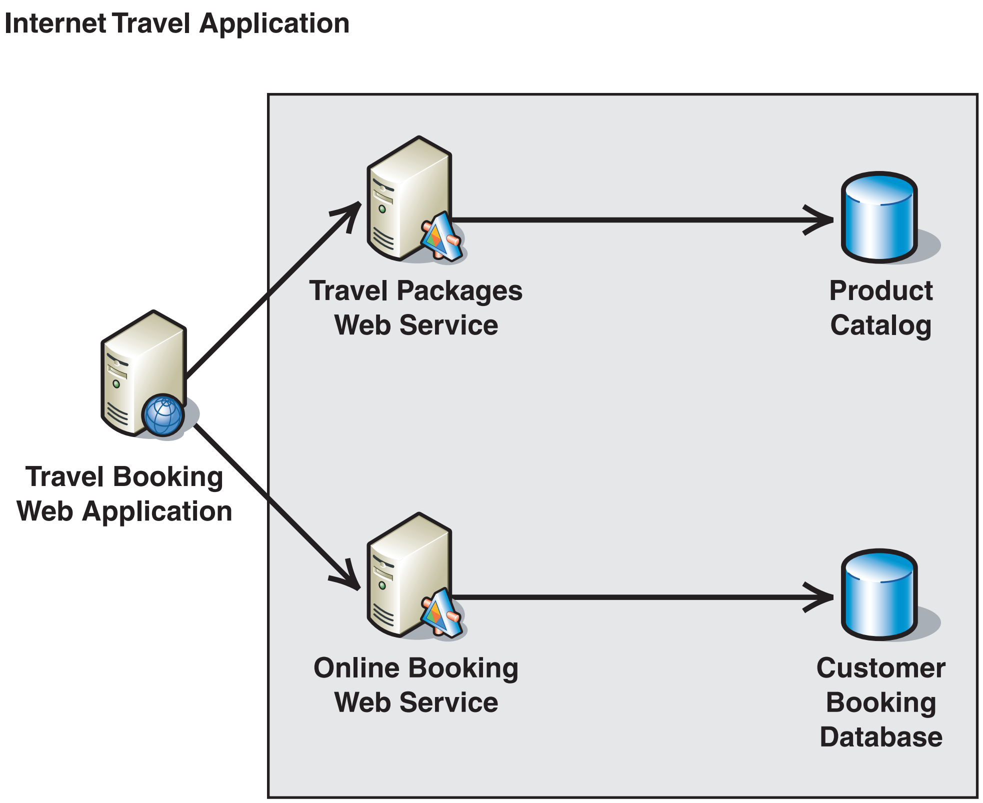

Internet Travel Application
Travel Packages Web Service
Product Catalog
Travel Booking Web Application
Online Booking Web Service
Customer Booking Database

Figure 8
An Internet-based travel booking application

The travel booking Web application is accessible from the Internet. However, only the Web application can access the Web services that the application calls. Each Web service has an independent data store. The following sections provide an overview of the travel booking application requirements.

Travel Booking Application Profile

The travel booking application has the following features:

- Travel agents in a travel franchise help customers book tour packages.
- Two Web services are used: a travel packages Web service, and an online booking Web service.
- The travel packages Web service provides travel product catalog information such as tour dates, itineraries, and prices.
- The online booking Web service allows travel agents to book tour packages on behalf of the customers.
- Identity propagation is needed for the online booking Web service because the database needs to keep a record of each travel agent who makes a travel request. Customers can go to any travel agent in the franchise to book a tour.
- During peak travel seasons, user activity is high. This means that performance must be considered.

Solution Approach

Table 7 lists factors considered for the Internet-based travel booking scenario and how each factor maps to a specific category.

Table 7: Internet Travel Booking Application Factors

Factor	Security consideration	Category
Security infrastructure	Travel agent user accounts are stored in a database.	Authentication
Security infrastructure	Servers used to host the Web services are behind a firewall.	Protection Scope
Security infrastructure	The travel agent franchise does not have a PKI.	Authentication
Organization security policies	Mutual authentication is required for all Web service interactions.	Authentication
Organization security policies	Applications must support SSO capabilities.	Authentication
Security treats	The online booking service handles sensitive data that must be protected against unauthorized access.	Message Protection

The information in Table 7 is combined with the business requirements related to security, and then it is grouped by category. Each category represents one or more decision matrices. The next step is to apply the security considerations to the appropriate matrices to make security decisions. In this example, you would examine the authentication decision matrices in Chapter 1, "Authentication Patterns," the message protection decision matrix in Chapter 2, "Message Protection Patterns," and the protection scope decision matrix in Chapter 3, "Implementing Transport and Message Layer Security."

Table 8 provides a summary of the decisions that were made after applying the security considerations from Table 7 and the related business requirements to the appropriate decision matrices.

Table 8: Summary of Security Decisions

Category	Security consideration	Security decision
Authentication	Travel agent user accounts are stored in a database.	When user credentials are stored in a database, Direct authentication is used to authenticate the user. To support SSO capabilities, Direct authentication can be combined with brokered authentication using a Security Token Service (STS).
Authentication	Mutual authentication is required.	The use of a shared symmetric key with STS provides mutual authentication.
Authentication	SSO support is required.	The security token issued by an STS can be used to access multiple Web services.
Authentication	Performance must be considered.	Brokered authentication speeds up operations when multiple Web services are accessed. In other words, authentication is performed only once. Encryption with a shared symmetric key is much faster than asymmetric methods. Only one of the Web services requires encryption.
Protection Level	Sensitive data must be protected against unauthorized access.	The security token issued by an STS can be used to provide data confidentiality and data origin authentication.
Protection Scope	Web services are behind a firewall.	Message layer protocols are easier to implement when passing through firewalls because additional ports do not need to be opened.

Solutions Description

This solution uses the following patterns to implement a combination of direct authentication and brokered authentication:

- Direct Authentication in Chapter 1, "Authentication Patterns."
- Brokered Authentication in Chapter 1, "Authentication Patterns."
- Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Brokered Authentication: Security Token Service (STS) in Chapter 1, "Authentication Patterns."
- Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
- Trusted Subsystem in Chapter 4, "Resource Access Patterns."

Direct authentication would be used to access a Security Token Service (STS) using WSE 3.0. The security token that is returned would then be used for brokered authentication against the Web services. The security token can also be used to provide data confidentiality and data origin authentication support as needed.

Figure 9 illustrates the security solution developed for the travel booking application.

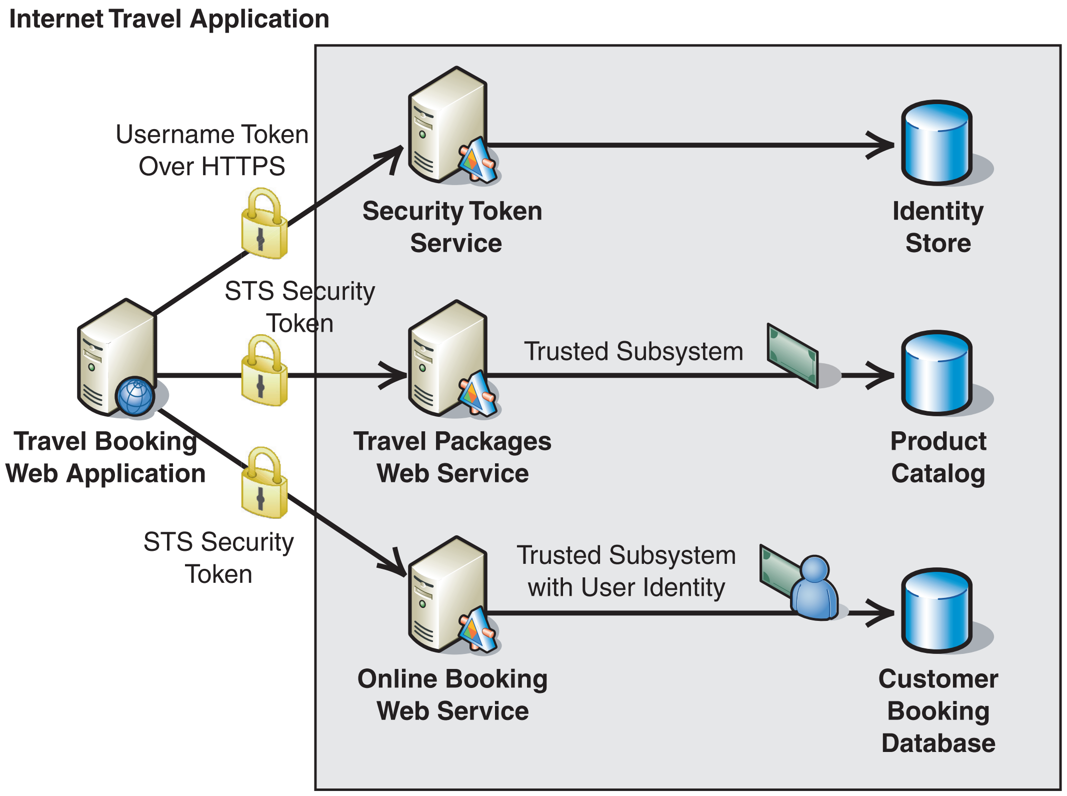

Internet Travel Application
Username Token Over HTTPS
Security Token Service
Identity Store
STS Security Token
Trusted Subsystem
Travel Booking Web Application
Travel Packages Web Service
Product Catalog
STS Security Token
Trusted Subsystem with User Identity
Online Booking Web Service
Customer Booking Database

Figure 9

The security solution for the Internet-based travel booking application

The Internet travel booking security solution is implemented in the following way:

- The STS uses a server certificate to establish secure communications with the travel booking Web application using HTTPS.
- The travel booking Web application passes a **UsernameToken** to the STS for authentication.
- The STS returns a security token for interaction with both the travel packages Web service and the online booking Web service.
- Encryption is not required when accessing the travel package Web service. However, the STS security token is used to sign the messages to provide authentication.
- The STS security token is used to sign and encrypt messages sent to the online booking Web service.
- A trusted subsystem is used to access the product catalog and customer booking database.
- Impersonation is not required for auditing. Instead, the agent's ID is retrieved from the security token and passed to the customer booking database as part of the request.

Additional patterns that could have been considered include:

- Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
- Message Validator in Chapter 5, "Service Boundary Protection Patterns."
- Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."

Part I

Core Web Service Security Patterns

In This Part:

1

Authentication Patterns

Introduction

As computer systems have increased in complexity, the challenge of authenticating users has also increased. As a result, there are a variety of models for authentication. For example, clients accessing a Web application may directly provide credentials, such as a user name and password for authentication. However, a third-party broker, such as a Kerberos domain controller, may be used to provide a security token for authentication. These two models are referred to as direct authentication and brokered authentication.

This chapter provides architectural patterns for direct authentication and brokered authentication, along with three brokered authentication design patterns that illustrate authentication using the Kerberos protocol, X.509, and a Security Token Service (STS). Figure 1.1 is a pattern map that illustrates how these patterns are related to one another.

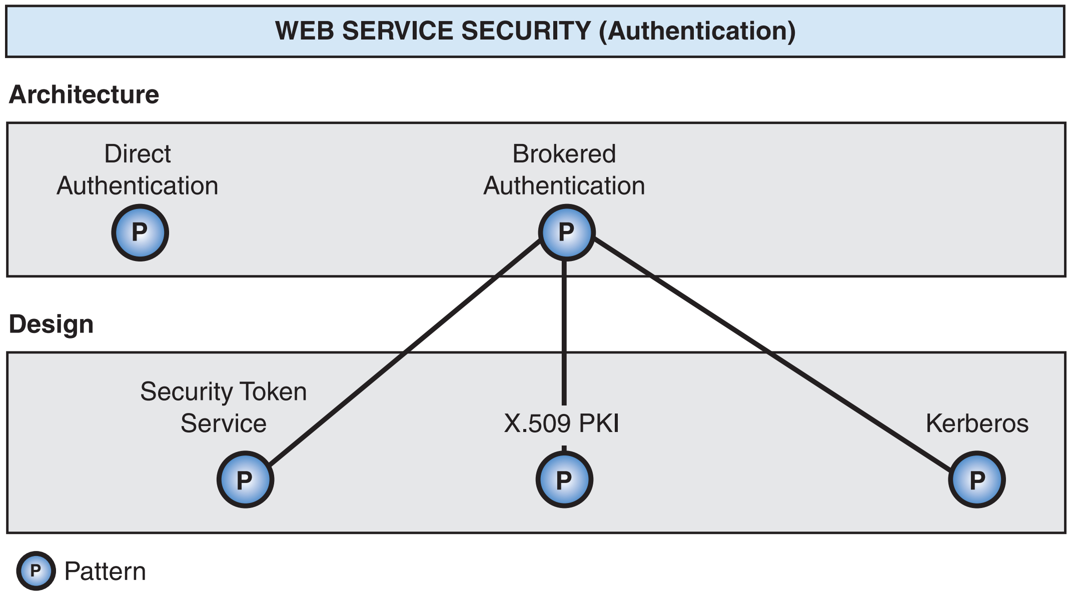


Figure 1.1
Authentication patterns

Authentication is considered to be a primary security feature because mechanisms used for authentication often influence mechanisms used for enabling other security features, such as data confidentiality and data origin authentication. For example, consider a case where the Kerberos protocol is used for message layer authentication. After a Kerberos session is set up between the client and service, it is possible to derive encryption keys from the Kerberos session key to encrypt application messages. From an architecture perspective, this is an advantage because you do not have to consider another security mechanism and infrastructure just to satisfy the data confidentiality needs.

Note: This introduction also discusses authorization, a concept that is intrinsically linked to authentication. However, the subject of authorization is already extensively documented, so the content in this chapter is intended to be only an introduction to the subject. An update to this guide is scheduled to coincide with the release of the Windows Communication Foundation (WCF). The update will incorporate patterns associated with distributed authorization.

Important Concepts

To fully understand authentication and authorization, it is important to understand the following concepts:

- **Authentication**. Authentication is the process of identifying an individual using the credentials of that individual. For example; with the driver's license example, a bank teller may be required to authenticate who you are by examining your driver's license. Authentication typically occurs immediately after identification.
- **Authorization**. Authorization is the process of determining whether an authenticated client is allowed to access a resource or perform a task within a security domain. Authorization uses information about a client's identity and/or roles to determine the resources or tasks that a client can perform.
- **Credentials**. A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential. A driver's license is an example of a credential in the real world. It contains data representing your identity and capabilities. It contains proof of possession in the form of your picture ID. It is issued by a trusted authority, such as your state department of licensing.
- **Identification**. Represents the use of an identifier that allows a system to recognize a particular subject and distinguish it from other users of the system.

Direct Authentication vs. Brokered Authentication

Both the Direct Authentication pattern and the Brokered Authentication pattern focus on the relationships that exist between a client and service participating in a Web service interaction. When both the client and service participate in a trust relationship that allows them to exchange and validate credentials including passwords, direct authentication can be performed, as shown in Figure 1.2.

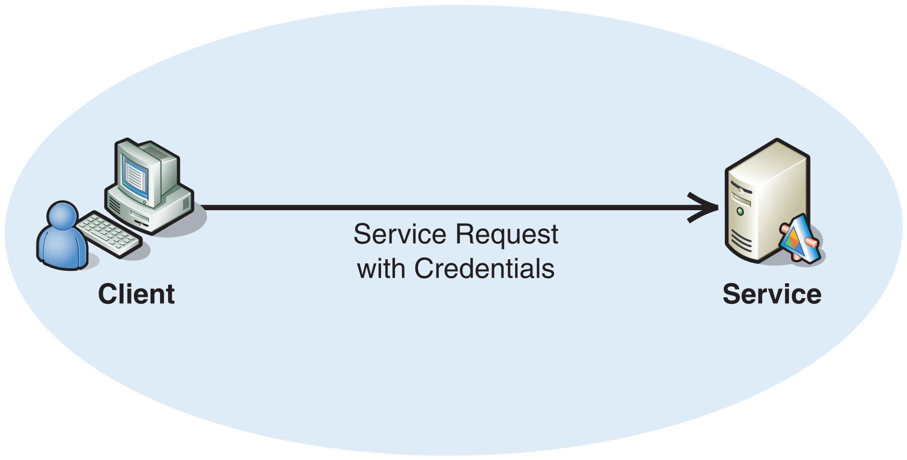


Figure 1.2
Direct authentication when a client and service share a trust relationship

An example of when this might be appropriate is where client applications and the service are able to establish credentials prior to the client using the Web service's capabilities. For example, before accessing a company's stock tracking service, you first establish an ID and password with the provider that you can then use to call its Web service. Another example is where the Web service wraps a legacy application that incorporates a custom authentication implementation that requires a user name and password to authenticate the client using information from a database.

In a situation where the client and service do not share a direct trust relationship, you can use a broker to perform authentication, as shown in Figure 1.3.

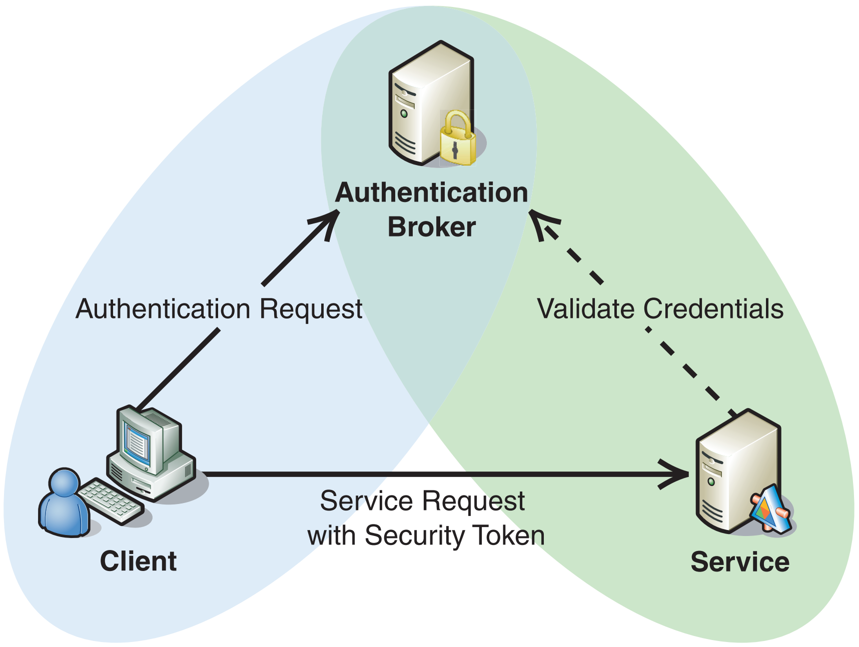


Figure 1.3
Using a broker to perform authentication when client and service do not share trust relationship

The broker authenticates the client and then issues a security token that the service can use to authenticate the client. The security token is always verified, but typically, the service does not need to interact with the broker to perform the verification. This is because the token itself can contain proof of a relationship with the broker, which can be used by the service to verify the token.

In addition to the different relationships, there are other security considerations that may support one approach over the other. For example, message protection requirements may dictate the use of brokered authentication when direct authentication is available. The support for different security infrastructures also has an influence on the authentication method used. Table 1.1 represents a decision matrix that lists security considerations related to authentication and how each one is supported by direct authentication methods or brokered authentication methods.

Table 1.1: Authentication Decision Matrix

Security Consideration	Direct	Brokered
What will the service require to prove the client's identity for authentication? Passwords, certificates, or something else?	Direct authentication requires the presentation of credentials, which are typically a user name and password. The service uses these credentials to authenticate the request.	Credentials are used to authenticate with the broker, which issues a security token. The security token is then used to authenticate with services.
Will the Web service be able to communicate with the authentication service that can validate the client's credentials for authentication?	Because authentication is performed, direct access to the authentication service is required.	Most implementations of brokered authentication do not require direct access to the authentication service.
Is there existing infrastructure to leverage?	Direct authentication works with any infrastructure used to provide credential management.	Brokered authentication requires an infrastructure in place that supports the type of security token that is used.
Is single sign on (SSO) support required?	Requires authentication for every call. The process of authenticating the client on every call can have a negative impact on performance.	Uses a security token that allows access to services after authentication has been performed. This same token could be used to access all services within an organization.
Will your application need to make multiple calls to the same service? In other words, should a security session be established?	It is possible to cache the user name and password; however, that is not a recommended procedure.	Security tokens can be used to establish a security session. Most tokens have a short lifetime and can safely be cached for multiple calls.
Is Windows impersonation or delegation required?	If a user name and password are sent with the message, it is possible to impersonate the client. This works only if the client has a Windows account.	The Kerberos protocol provides the ability to implement delegation.

One thing to keep in mind is that these are just some of the security considerations that need to be examined. Other considerations related to security policies and threats identified during a security analysis should also be examined.

Brokered Authentication Options

The three main security tokens provided by WSE support brokered authentication. These tokens are X.509, **KerberosToken**, and a custom security token issued by a Security Token Service (STS).

Table 1.2 represents a decision matrix that lists security considerations related to authentication and how each one is supported by different security tokens.

Table 1.2: Security Token Decision Matrix

Security Consideration	X.509	KerberosToken	Custom (STS)
Existing infrastructure	Requires support for a Public Key Infrastructure (PKI), which can be expensive to set up and maintain. In cases where a limited number of certificates are needed, an external certificate authority (CA) can be used.	Requires an identity provider that supports the Kerberos protocol, such as Active Directory.	Requires an STS implementation that issues and manages security tokens.
The client and the service reside within the same organizational boundary	X.509 certificates can be used across organizational boundaries. Management of certificates can become difficult with a large number of partners.	The Kerberos protocol is used to authenticate clients within a domain. Cross-domain trusts can be established but are typically limited within an organization.	A custom STS can provide authentication across organizational boundaries if both parties can standardize on the verification and processing of the token.
Support for Windows impersonation or delegation	Can be used for impersonation when a certificate is mapped to a client within Active Directory.	Supports both impersonation and delegation.	Not supported.
Support for security sessions	Most X.509 implementations, such as SSL, exchange a symmetric session key that is used for encryption.	Service tickets are session-based tokens that can be used for confidentiality and integrity.	Custom security tokens can be used for session based operations.

(continued)

Table 1.2: Security Token Decision Matrix *(continued)*

Security Consideration	X.509	KerberosToken	Custom (STS)
Interoperability with other platforms or technologies	Based on industry standards and supported on many platforms. Often used for interoperability with Java.	Based on industry standards, with availability on most major platforms; however, adoption is probably not as extensive as X.509.	Based on the implementation of the STS.
Support for message protection	Can be used to provide confidentiality and data origin authentication at the message layer and transport layer.	Supports confidentiality and data origin authentication at the message layer and supports transport layer when used with IPSec.	Supports confidentiality and data origin authentication at the message layer only.
Your application has a requirement to support non-repudiation and auditing	Signatures created using X.509 certificates can be mapped to a particular participant in a conversation — assuming both participants have unique certificates. The identity of a particular client can be mapped to a certificate.	Kerberos tokens can be used for impersonation and delegation, which makes them the ideal choice for auditing.	Based on the implementation of the STS.

Keep in mind that these are some of the main security considerations. Other considerations related to security policies or a threat analysis will also influence your choice.

Authorization Methods

The .NET Framework currently supports two different methods for performing authorization, role-based authorization and resource-based authorization.

Role-Based Authorization

Role-based authorization is used to associate clients and groups with the permissions that they need to perform particular functions or access resources. When a user or group is added to a role, the user or group automatically inherits the various security permissions. Role-based authorization can be declarative or imperative.

Note: Authorization can be based on any attribute of a security principal. However, the majority of non-resource–based authorization methods use roles for authorization purposes.

Declarative

Declarative role-based authorization can be added to application code at design time. Required access for a particular method or class is declared as an attribute in code. Attribute metadata is discoverable using reflection; this makes it easier to track at design time the security principals that are allowed to access the method. The following code is an example of declarative security in .NET that uses an attribute on a method to require that the current principal on the thread belongs to the Administrators group.

```
using System.Security.Permissions;
...
 [PrincipalPermissionAttribute(SecurityAction.Demand, Role = "Administrators")]
 public static void PrivateInfo()
 {
    //Print secret data.
    Console.WriteLine("\n\nYou have access to the private data!");
 }
```

Imperative

Imperative role-based authorization is written into the application code to make authorization decisions at run time. Imperative security is useful when the resource to be accessed or action to be performed is not known until run time or when finer-grained access control beyond the level of a code method is required. The following code is an example of imperative security in .NET that checks role membership at run time.

```
using System.Security.Principal;
using System.Threading;
...

WindowsPrincipal MyPrincipal = (Thread.CurrentPrincipal as WindowsPrincipal);
if (MyPrincipal.IsInRole("Administrator"))
{
    // Permit access to some code.
}
```

Unlike declarative security, role checking when using imperative security does not have to be performed on the application thread's current principal. A reference to a security principal can be obtained at run time to check roles for a user that may not be currently logged into the application.

Resource-based Authorization

Resource-based authorization is performed declaratively on a resource, depending on the type of the resource and the mechanism used to perform authorization. Resource-based authorization can be based on access control lists (ACLs) or URLs.

Access Control List (ACL)

Individual resources, such as files, are secured using Windows ACLs that specify the type of operation a particular security principal or group to which a security principal belongs can perform on the object. The application impersonates the caller prior to accessing resources; because of this, the operating system can perform standard access checks. All resource access is performed using the original caller's security context.

URL Authorization

URLs are declared with permissions for the URL to define who is authorized to access the URL in question. Typically, this approach is linked to an authorization store such as Authorization Manager (AzMan), which defines access entitlements for a specific URL and maps those entitlements to user logons.

Policy

Policy provides a means to declaratively enforce security on SOAP request and response messages through policy assertions. The policy implementations in WSE are based on the WS-SecurityPolicy and WS-PolicyAssertions specifications. Policies for an application are stored in a cache file that is referenced in the WSE configuration section of the application's configuration file. A policy consists of one or more assertions that express a security requirement, capability, or preference for an inbound or outbound SOAP message. Required claims on a security token attached to a message can be used to authorize access to a Web service or a specific operation on a Web service.

WSE policy is used to provide confidentiality, integrity, and data origin authentication, as shown in the following patterns in Chapter 3, "Implementing Transport and Message Layer Security":

- Implementing Direct Authentication with UsernameToken in WSE 3.0
- Implementing Message Layer Security with X.509 Certificates in WSE 3.0
- Implementing Message Layer Security with Kerberos in WSE 3.0

Authorization in Windows Communication Foundation

Windows Communication Foundation (WCF) will integrate seamlessly with the role-based security features that are built into the .NET Framework. WCF will communicate the sender's credentials to the receiving code using the usual **Thread.CurrentPrincipal** property. Because of this, you can perform authorization either declaratively using **PrincipalPermissionAttribute** or imperatively using **IPrincipal.IsInRole**.

WCF will also incorporate a sophisticated claims-based authorization infrastructure exposed through an object named "authorization context." This allows more complex authorization decisions to be made based on additional claims provided in tokens within an incoming message.

Guidance for authorization using WCF will be incorporated in an updated version of this guidance that will also incorporate implementations using WCF.

For a complete description of authorization on the .NET Framework, see Authentication and Authorization on MSDN®.

The remainder of this chapter contains the architecture and design patterns related to authentication. The architecture patterns are the following:

- Direct Authentication
- Brokered Authentication

The design patterns are the following:

- Brokered Authentication: Kerberos
- Brokered Authentication: X.509 PKI
- Brokered Authentication: Security Token Service (STS)

Direct Authentication

Context

A client needs to access a Web service. The Web service requires the client to present credentials for authentication so that additional controls such as authorization and auditing can be implemented.

Problem

How does the Web service verify the credentials that are presented by the client?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **The credentials that the client presents to the Web service are based on shared secrets, such as passwords**. Authentication of individual users is often performed with passwords. Computers and applications often use higher quality secrets that are more secure than passwords. The client and the Web service must exchange the shared secrets securely before interaction is possible. The exchange of shared secrets must occur through an out-of-band mechanism.
- **The Web service can validate credentials from the client against an identity store**. The Web service must have direct access to the identity store, including appropriate permissions for accessing identity information.
- **The Web service is relatively simple, and does not require support for capabilities such as single-sign on (SSO) or support for non-repudiation**. In these circumstances an effective, low cost solution that does not use an authentication broker may be possible.
- **The client and the Web service trust one another to manage credentials securely**. In this situation, both parties should consider the credentials as equal in value to the information and services they protect. If either the Web service or the client manage the credentials in an insecure manner, neither party can be sure that the mishandled credentials prove the identity of the user or application.

Solution

Use direct authentication where the Web service acts as an authentication service to validate credentials from the client. The credentials, which include proof-of-possession that is based on shared secrets, are verified against an identity store.

Participants

Direct authentication involves the following participants:

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.
- **Identity store**. The entity that stores a client's credentials for a particular identity domain.

Process

Figure 1.4 identifies the tasks that occur during direct authentication.

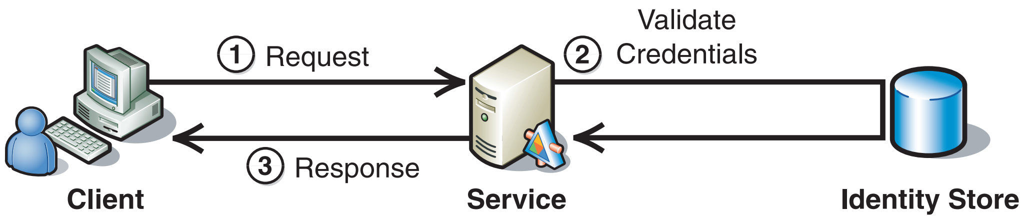


Figure 1.4
The direct authentication process

As illustrated in Figure 1.4, the following steps describe the direct authentication process:

1. The client sends a request to the Web service, attaching credentials to the request message.
2. The Web service validates the credentials against an identity store and makes authorization decisions about the client.
3. The Web service returns a response to the client. (This step is optional.)

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Direct Authentication pattern include the following:

- It represents an uncomplicated model for authenticating clients without the need for an authentication broker.
- If the shared secret between a requester and service is compromised, only the relationship between those two parties is compromised and not the entire model.

Liabilities

The liabilities associated with the Direct Authentication pattern include the following:

- Direct authentication does not provide single sign on capabilities. Without single sign on, the client may be forced to authenticate prior to every Web service call or to cache the user's credentials within the application. If the user's credentials include a password, caching the password is not recommended because it may pose a security risk.
- The decentralized nature of direct authentication requires that the trust relationship be managed between each point in the communication, as shown in Figure 1.5.

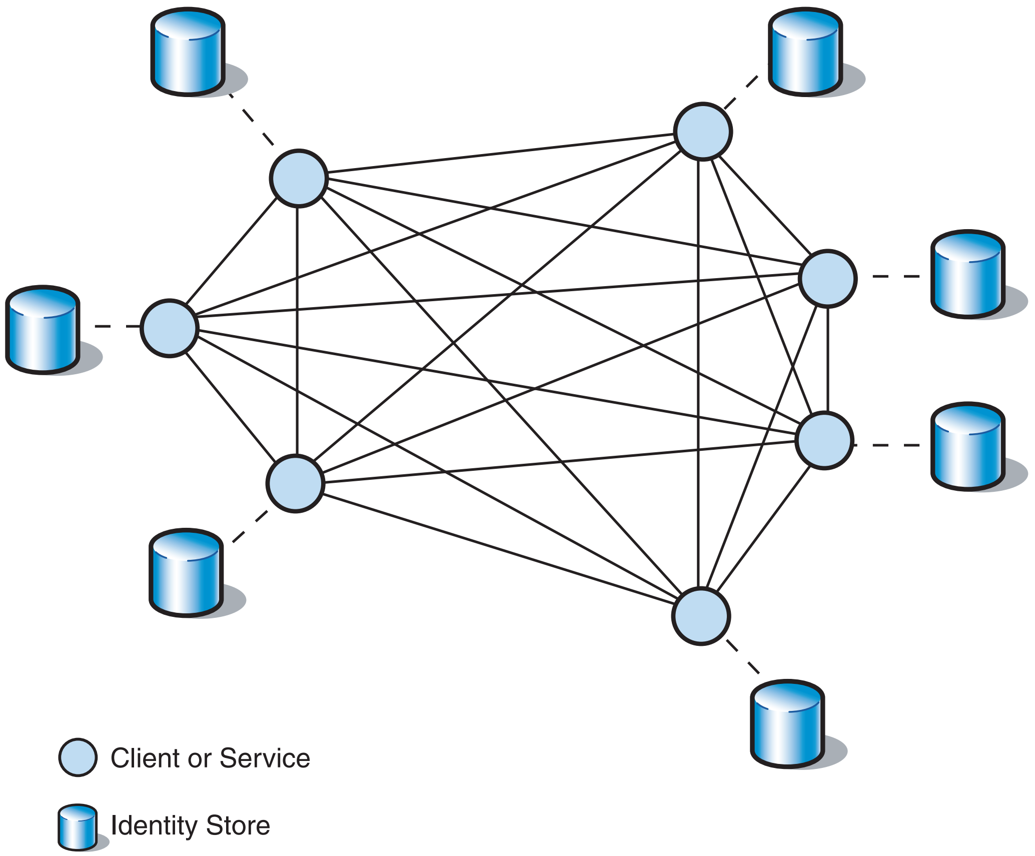


Figure 1.5
Trust relationships between points of communication in direct authentication

Each line in Figure 1.5 represents a discreet trust relationship established by authentication with a shared secret. As the number of discreet relationships between clients and services increases, each with potentially different identity stores, the challenges of managing and distributing secrets becomes more complicated.

- If a client calls a Web service frequently, the use of direct authentication can increase latency, because the Web service typically authenticates against a remote identity store.
- Data ownership and synchronization issues can occur if each of several services has its own identity store to authenticate the same client. This is because the client's credentials may need to be duplicated across multiple identity stores.

Security Considerations

Security considerations associated with the Direct Authentication pattern include the following:

- An attacker can impersonate the client if he or she intercepts the client's shared secret. The identity secret may be obtained if it is unprotected in transit or successfully guessed offline. You should use encryption to provide data confidentiality for this data. For more information, see Data Confidentiality in Chapter 2, "Message Protection Patterns."
- A shared secret is sensitive data and must be secured whenever it is persisted — even if it will be held for only a short time in a message queue. Shared secrets must be protected when stored in an identity store. If an attacker gains unauthorized access to an identity store that stores passwords in plaintext, all passwords in the identity store are immediately compromised. This allows the attacker to impersonate any user. Most authentication services such as Active Directory and Lightweight Directory Access Protocol (LDAP)-enabled directory services use identity stores that store passwords as either hashed, encrypted, or both. However, if you implement a custom identity store such as a database, you must ensure that the passwords are protected. Although a brute force or pre-computed dictionary attack is possible against a hashed password, hashing the passwords in the database will protect them from immediate disclosure in the event that an attacker gains access to them. For more information about how to hash passwords in a database, see Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
- If a client calls a Web service after a user has authenticated, it must cache the username and password locally for presentation on subsequent calls to the Web service for direct authentication. Caching secrets, such as passwords, increases the risk of disclosure if an attacker is able to gain access to the cache or flush the contents of the cache to an accessible location. You should secure the cache mechanism so that its confidentiality and integrity can be maintained to prevent disclosure or tampering. If the client is a Web application in a Web farm, you may want to consider brokered authentication instead of direct authentication. Otherwise, the user may be forced to re-authenticate if a request is routed to a different server in the farm after the user has authenticated.

Related Patterns

Three types of patterns are related to this pattern: child patterns, alternate patterns, and additional patterns.

The following child patterns are related to the Data Authentication pattern:

- **Implementing Direct Authentication with UsernameToken in WSE 3.0**. This implementation pattern focuses on using direct authentication at the message layer.
- **Implementing Transport Layer Security Using HTTP Basic over HTTPS**. This reference provides information about implementing direct authentication using Internet Information Services (IIS) with X.509 certificates at the transport layer.

The following alternate pattern is related to the Direct Authentication pattern:

- **Brokered Authentication**. This pattern is an alternative to direct authentication that describes how to prove a client's identity to an authentication broker for issuance of a security token, and then use the issued security token to authenticate with a service.

The following pattern may use the Direct Authentication pattern:

- **Brokered Authentication**. This pattern may use variations of direct authentication to prove a client's identity to an authentication broker for issuance of a security token.

Brokered Authentication

Context

A client needs to access a Web service. The Web service requires the application to present credentials for authentication so that additional controls such as authorization and auditing can be implemented.

Problem

How does the Web service verify the credentials that are presented by the client?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **The client accesses additional services, which results in the need for a single sign on (SSO) solution**. Without a single sign on solution, the client may be forced to authenticate prior to every Web service call or cache the user's credentials within the application. If the user's credentials include a password, caching the password is not recommended because it may pose a security risk.
- **The client and the Web service do not trust each other directly**. The client and the Web service may not trust one another to manage or exchange shared secrets securely. Establishing trust directly between a client and Web service often requires out of band interactions that can hinder clients and services from interacting dynamically.
- **The Web service and the identity store do not trust each other directly**. The Web service may be unable to communicate with the identity store directly, because of access control restrictions, network restrictions, or organizational policy.

The following condition is an additional reason to use the solution.

- **The client and Web service share a standard access control infrastructure**. You can simplify the development of new Web services by standardizing and centralizing the issuance and verification of credentials. You can also centralize the management of data associated with credentials; this reduces the costs associated with identity management.

Solution

Use brokered authentication where the Web service validates the credentials presented by the client, without the need for a direct relationship between the two parties. An authentication broker that both parties trust independently issues a security token to the client. The client can then present credentials, including the security token, to the Web service.

Participants

Brokered authentication involves the following participants:

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.
- **Authentication broker**. The authentication broker authenticates clients and maintains authoritative control over security tokens. It also vouches for the client by issuing it a security token.
- **Identity store**. The entity that stores a client's credentials for a particular identity domain.

Process

Figure 1.6 depicts the interactions that are performed during brokered authentication.

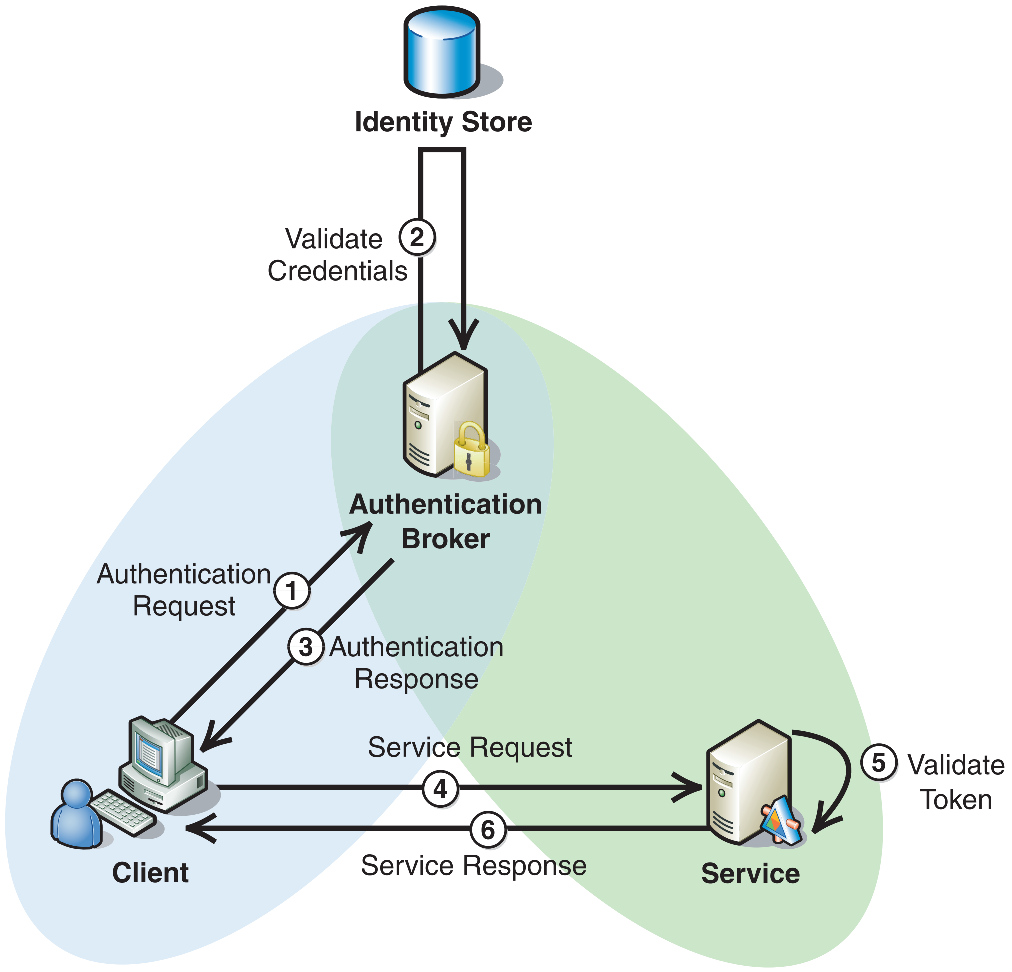


Figure 1.6
Brokered authentication process

As illustrated in Figure 1.6, the following steps describe the brokered authentication process:

1. The client submits an authentication request to the authentication broker.
2. The authentication broker contacts the identity store to validate the client's credentials.
3. The authentication broker responds to the client, and if authentication is successful, it issues a security token. The client can use the security token to authenticate with the service. The security token can be used by the client for a period of time that is defined by the authentication broker. The client can then use the issued security token to authenticate requests to the service throughout the lifetime of the token.
4. A request message is sent to the service; it contains the security token that is issued by the authentication broker.
5. The service authenticates the request by validating the security token that was sent with the message.
6. The service returns the response to the client.

There are different types of authentication brokers. Each type uses different mechanisms to broker authentication between a client and a service. Common examples of an authentication broker include the following:

- X.509 PKI
- Kerberos protocol
- Web Service Security Token Service (STS)

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Brokered Authentication pattern include the following:

- The authentication broker manages trust centrally. This eliminates the need for each client and service to independently manage their own trust relationships, as shown in Figure 1.7.

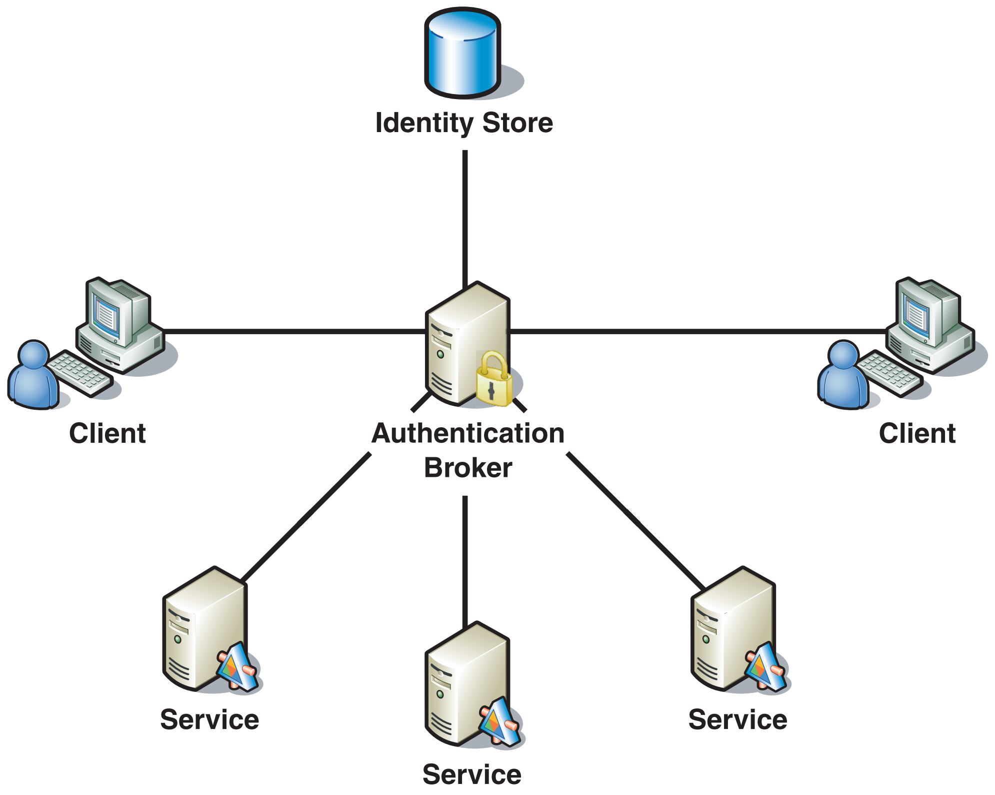


Figure 1.7
An authentication broker centrally managing trust

- Solutions built around brokered authentication with a centralized identity provider are often easier to maintain than direct authentication solutions. When new users who require access to any of the clients or Web services are added to the identity store, their credentials are maintained in one central point.
- Two parties participating in brokered authentication do not require prior knowledge of one another to communicate. If a client is modified to call a Web service it has never used before, the Web service requires no changes to its configuration or data to authenticate credentials presented by the client.
- Trust relationships can be established between different authentication brokers. This means that an authentication broker can issue security tokens that are used across organizational boundaries and autonomous security domains.

Liabilities

The liabilities associated with the Brokered Authentication pattern include the following:

- The centralized trust model that is used by Brokered authentication can sometimes create a single point of failure. Some types of authentication brokers, such as the Kerberos Key Distribution Center (KDC), must be online and available to issue a security token to a client. If the authentication broker somehow becomes unavailable, none of the parties that rely on the authentication broker to issue security tokens can communicate with each other. This problem of a single point of failure can be mitigated by implementing redundant or back-up authentication brokers, although this increases the complexity of the solution.
- Any compromise of an authentication broker results in the integrity of the trust that is provided by the broker also being compromised. If an attacker does successfully compromise the authentication broker, it can use the authentication broker to issue security tokens, and conduct malicious activity against parties that trust the authentication broker.

Security Considerations

Security considerations associated with the Brokered Authentication pattern include the following:

- Claims held in security tokens often contain sensitive data, and must be protected in transit, either by using message layer security, or transport level security.
- Security tokens must be signed by the issuing authentication broker. If they are not, their integrity cannot be verified. This could result in attackers trying to issue false tokens.
- A Time of Change/Time of Use vulnerability may exist if the client's account status, identity attributes, or authorization attributes are modified by an account administrator. If these changes are not reflected in the security token, it creates a vulnerability that may lead to invalid clients interacting with the service with elevated privileges.

Related Patterns

Three types of patterns are related this pattern: child patterns, alternate patterns, and additional patterns.

The following child patterns are related to the Brokered Authentication pattern:

- **Brokered Authentication: X.509 PKI**. This pattern describes a specialized authentication broker based on the X.509 PKI standard.
- **Brokered Authentication: Kerberos**. This pattern describes a specialized authentication broker based on the Kerberos authentication protocol.
- **Brokered Authentication: Security Token Service (STS)**. This pattern describes a specialized authentication broker in the form of a Security Token Service.

The following alternate pattern is related to the Brokered Authentication pattern:

- Direct Authentication. This pattern is an alternative to brokered authentication where authentication is based on an identifier and a shared secret, such as username and password.

One additional pattern is related to the Brokered Authentication pattern:

- Broker. This pattern is in *Enterprise Solution Patterns Using Microsoft .NET* on the MSDN Web site. This pattern shows how to hide the implementation details of remote service invocation.

Brokered Authentication: Kerberos

Context

Web services must authenticate clients so that additional controls, such as authorization and auditing, can be implemented. The organization has decided to use an *authentication broker* to provide a common access control infrastructure for a group of applications. The authentication broker negotiates trust between client applications and Web services, which removes the need for a direct relationship. The authentication broker should issue signed security tokens that can be used for authentication.

Problem

How does the Web service verify the credentials presented by the client application?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Users access multiple clients that call Web services, resulting in the need for single sign on (SSO) capabilities**. To ensure a good user experience, users should only have to enter a username and password when they logon to their workstations. They should not need to re-enter them multiple times to access multiple clients.
- **Centralized authentication of clients is required**. Management of user and computer credentials must be centralized to minimize security risks associated with persisting credentials and to reduce maintenance overhead.
- **Clients that require authentication are implemented on a variety of platforms within the organization, and the organization has identified a need for interoperability between those platforms**. The easiest way to attain interoperability between different platforms is to use a standards-based mechanism for authentication.

- **Clients may exist in an untrusted network environment**. You may not be able to guarantee the security of the computers on the network or of the network itself. User credentials must be protected from malicious attackers that may have gained access to the network inappropriately.

The following condition is an additional reason to use the solution:

- **Applications require some of the extended capabilities associated with a particular implementation of the Kerberos protocol**. For example, the Windows Server 2003 implementation of the Kerberos protocol provides capabilities, such as protocol transition, constrained delegation, and integration with Active Directory.

Solution

Use the Kerberos protocol to broker authentication between clients and Web services.

The client requests a ticket from an authentication broker, which returns a service ticket and session key used to create a Kerberos security token. The security token includes the service ticket and a data structure called an *authenticator*, which is encrypted by using the session key retrieved from the broker. Then the Kerberos security token is sent with a request message to the Web service.

When the Web service receives a Kerberos security token, it extracts the service ticket and uses a long-term service key to decrypt the service ticket. The Web service uses the session key from the service ticket to decrypt the authenticator and authenticate the client.

Note: The term *Kerberos security token* is used to represent a data structure that contains a service ticket and authenticator. For more information on Kerberos tickets and authenticators, see Kerberos Technical Supplement for Windows in Chapter 7, “Technical Supplements.”

Participants

Brokered authentication using the Kerberos protocol involves the following participants:

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.
- **Key Distribution Center (KDC)**. The KDC is the authentication broker that is responsible for authenticating clients and issuing service tickets. On the Windows platform, the KDC is implemented in Active Directory.

The Kerberos protocol is an authentication protocol that requires the following additional components.

- **Account database**. This is an identity store that the Kerberos KDC uses to check client credentials presented for authentication. Master keys for the client and service are also stored in this database. If the Kerberos protocol is implemented on a Windows Server 2003 domain controller or on a Windows 2000 domain controller, then Active Directory provides this function.

Note: The term *master key* refers to a long-term key, which is described in the Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

- **Kerberos policy**. This is a security policy that defines behavior for a Kerberos realm, which is also an Active Directory domain. Policy settings include user logon restrictions, service ticket lifetime, user ticket lifetime, and clock synchronization.

Note: There is some inconsistency in how service tickets are described in Kerberos documents. The names *service ticket* and *session ticket* are used interchangeably. When you encounter the phrase *session ticket*, remember that this is a *service ticket*.

The relationship between participants is shown in Figure 1.8.

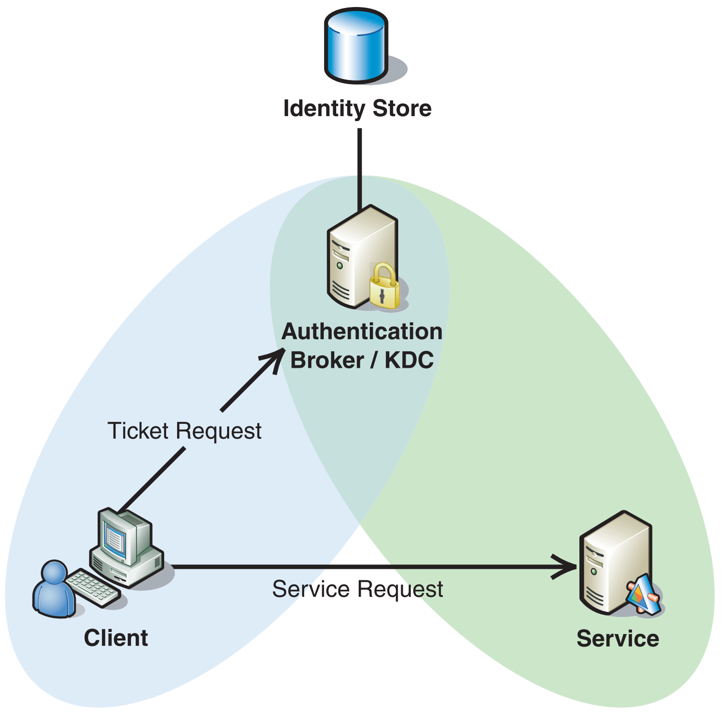


Figure 1.8
Relationship between participants

Process

Brokered authentication with the Kerberos protocol consists of the following high-level tasks:

1. **The client authenticates with the broker (KDC)**. The client is authenticated with the broker, and given access to a ticket-granting ticket (TGT) that can be used to request access to a service.
2. **The client authenticates with the service**. The client uses the TGT to request access to a particular service, and then it receives a service ticket. The service uses the service ticket to validate credentials.

Note: The Windows implementation of the Kerberos protocol uses many components and interfaces that are beyond the scope of this pattern. The process described in this pattern focuses on the interaction of primary components that the Kerberos protocol uses to authenticate with a Web service, and not on the low-level implementation of the Kerberos protocol on the Windows platform.

Client Authenticates with Broker (KDC)

Clients can be authenticated through a wide variety of techniques, including:

- Workstation user login using the secure attention sequence (CTRL+ALT+DELETE).
- Windows integrated authentication used to access a Web application.
- IIS process identity authentication used when the process starts.
- Protocol transition used to transition clients authenticated using a non-Windows protocol into a Kerberos security context.

The actual process of authenticating a client is beyond the scope of this pattern. However, it's important to understand that the client must first be authenticated with the broker and have access to a TGT before it can request access to a service.

For detailed information about the process of authenticating clients and issuing ticket-granting tickets on the Windows platform, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

For information about protocol transition, see Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."

Client Authenticates with Service

The process of authenticating with a service is shown in Figure 1.9.

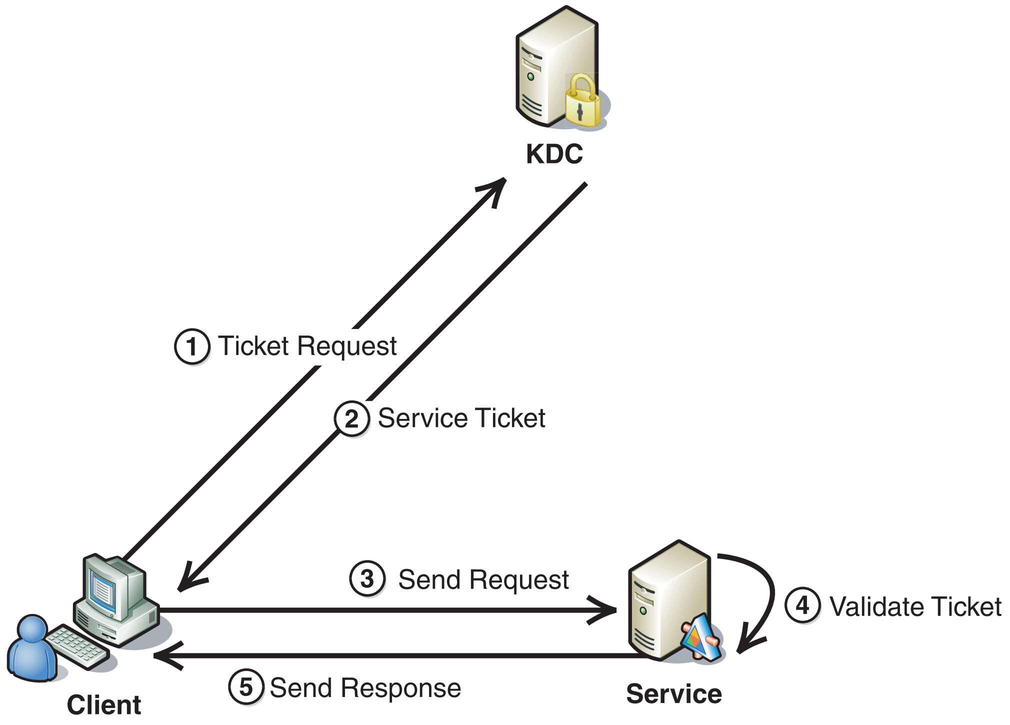


Figure 1.9
Service Authentication

The following steps describe the process of service authentication depicted in Figure 1.9:

1. The client sends a TGT in a message to the KDC to request a service ticket for communication with a specific Web service.
2. The KDC creates a new session key and service ticket that will be used for communication with the requested service. The service ticket contains the client's authorization data and the new session key. The KDC encrypts the service ticket with the Web service's master key. The service ticket and encrypted session key are returned to the client.

   Both the new session key and service ticket represent credentials used to create a security token that allows access to the Web service. The client decrypts the session key and then uses the key to encrypt an authenticator, which contains a timestamp and other information. The authenticator and session ticket are included in the new Kerberos security token. The session key is not included in the token; however, it is included in the service ticket, which is what the service uses to validate the token.

3. A request message, which contains the Kerberos security token created in the previous step, is sent to the service.
4. The service uses its master key to decrypt the service ticket found in the security token and to retrieve the session key. The session key is used to decrypt the authenticator and validate the security token. When it is validated, the service accepts the security token and uses it to initialize a security context based on the client information contained in the service ticket.
5. (optional)The service returns a response to the client. To provide mutual authentication, the response should contain unique information that is encrypted with the session key to prove to the client that the service knows the session key.

The Kerberos protocol follows the basic pattern of brokered authentication, but it has properties that differentiate it from other types of brokered authentication, including the following:

- The Kerberos protocol supports the notion of ticket renewal, but it does not automatically revoke tickets. By default, Kerberos tickets have a fixed lifetime of 8 hours; however, the Windows implementation uses a fixed lifetime of 10 hours.
- The KDC does not terminate a service ticket when an authenticated client is finished communicating with a service. Instead, it lets the ticket expire at the end of its normal lifetime. Because tickets are used for authentication, if the ticket expires during communication with a service, the expiration will not affect current operations. Clients are not notified when a ticket is about to expire.

For more information on Kerberos tickets and ticket lifetime, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

The Kerberos protocol can be used for brokering authentication at either the transport layer or message layer. Some implementations of Kerberos authentication include the following:

- Transport-layer Kerberos authentication, which includes:
  - Windows Integrated Security
  - IP Security Protocol with Internet Key Exchange (IPSec/IKE)
- Message-layer Kerberos authentication, which includes:
  - Web Service Enhancements (WSE) 2.0 KerberosToken2
  - Web Service Enhancements (WSE) 3.0 KerberosToken

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Brokered Authentication: Kerberos pattern include:

- The Kerberos protocol provides SSO capabilities, which allow a client to authenticate only once per logon session.
- The Kerberos protocol has broad acceptance as a brokered authentication protocol and is in use in the majority of large organizations that have a centralized authentication infrastructure.
- The Kerberos protocol is closely integrated with the Windows operating system (Windows 2000 and later). This enables the operating system to provide additional capabilities, such as user impersonation/delegation, authorization, and auditing.
- Kerberos supports mutual authentication when a service sends a response that contains data encrypted with the shared session key.

Liabilities

The liabilities associated with the Brokered Authentication: Kerberos pattern include:

- The centralized nature of the Kerberos protocol requires a KDC, which acts as an authentication broker, to be available at all times. If the KDC fails, clients will not be able to establish new trust relationships with a service. You should consider using redundant KDCs or providing an alternative mechanism, such as X.509 certificates, for authentication. With Active Directory, KDC availability can be improved by establishing secondary domain controllers. This creates a redundant set of Kerberos KDCs.
- The Kerberos protocol is only useful for online authentication and secure communication. Kerberos is not useful for long-term persistence because of the limited lifetime of tickets and session keys used for encryption and signing.
- The Kerberos protocol cannot establish proof of authentication for a client outside of its security realm (Active Directory domain) unless a trust relationship has been established with the other security realm.

Security Considerations

Security considerations associated with the Brokered Authentication: Kerberos pattern include:

- Clients must keep their master keys secret. If an intruder somehow compromises a client's key, it will be able to masquerade as that client or impersonate any server to the legitimate client.
- With the Kerberos protocol, password guessing attacks can occur against messages encrypted with a password equivalent derived from the client's password. (For client authentication, this is the client's password. For service authentication, this is the password of the service account.) The Kerberos protocol uses this derived key to encrypt data in the authentication request. To discover the password, an attacker could mount an offline dictionary attack by repeatedly attempting to decrypt the data in the authentication request sent to the KDC.

- The Kerberos protocol does not implement authorization, although it is typically coupled with an identity store that may store authorization information for a client. Resources may control access based on the client's authorization information, which is contained in the service ticket.
- The Kerberos protocol cannot be used for non-repudiation because the client's identity secret is shared with the KDC.
- Each host on the network must have a clock that is loosely synchronized to the time of the other hosts. This synchronization reduces the bookkeeping needs of application servers when they do replay detection. You can configure the degree of looseness on a per-server basis. If the clocks are synchronized over the network, the clock synchronization protocol itself must be secured from network attackers.

Related Patterns

Three types of patterns are related to this pattern: parent patterns, child patterns and alternate patterns.

The following parent pattern is related to the Brokered Authentication: Kerberos pattern:

- **Brokered Authentication**. This pattern describes how to prove a client's identity to an authentication broker so that the broker can issue a security token.

The following child patterns are related to the Brokered Authentication: Kerberos pattern:

- **Implementing Brokered Authentication Using Windows Integrated Security on IIS**. This reference provides a concise reference on how to use Windows Integrated Security on IIS.
- **Implementing Message Layer Security with Kerberos in WSE 3.0**. This pattern provides implementation guidelines for using the Kerberos protocol in WSE 3.0 to implement brokered authentication, authorization, data integrity, and data origin authentication.
- **Protocol Transition with Constrained Delegation Technical Supplement**. This technical supplement describes different scenarios for using protocol transition, and then provides step-by-step details for implementing protocol transition. In addition, this pattern describes how a protocol transition can be used with constrained delegation to access downstream resources.

The following alternate patterns are related to the Brokered Authentication: Kerberos pattern:

- **Brokered Authentication: X.509 PKI**. This pattern describes a specialized authentication broker based on the X.509 PKI standard.
- **Brokered Authentication: Security Token Service (STS)**. This pattern describes a specialized authentication broker based on using a security token service.

Brokered Authentication: X.509 PKI

Context

Web services must authenticate clients so that additional controls, such as authorization and auditing, can be implemented. The organization has decided to use *brokered authentication*, based on the need for a single sign on (SSO) solution and to allow multiple Web services to share a standard access control infrastructure. The authentication broker should issue signed security tokens that can be used for authentication.

Problem

How does the Web service verify the credentials presented by the client?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **The environment includes multiple organizational boundaries or autonomous security domains**. The authentication broker must be able to issue security tokens that can be used across organizational boundaries.
- **The client and the Web service do not trust each other**. The client and the Web service may not trust one another to manage or exchange shared secrets securely. Establishing trust directly between a client and Web service could require offline interactions that can hinder clients and services from interacting dynamically.
- **The authentication broker might be offline or unavailable on some occasions**. The Web service must be able to validate authentication credentials when the authentication broker is not available. This ensures that the Web service can continue to process requests, even if the authentication broker becomes unavailable.
- **Clients that require authentication are implemented on a variety of platforms within the organization, and interoperability is required between those platforms**. Using a standards-based mechanism for authentication helps ensure interoperability between different platforms.
- **The organization may need to trace particular actions to a specific client or service**. A record of transactions allows an organization to provide evidence that a particular action was requested and/or performed. This could be useful if a user denies that he or she performed an action or if a client needs to verify that a service has performed a specific task.

Solution

Use brokered authentication with X.509 certificates issued by a certificate authority (CA) in a public key infrastructure (PKI) to verify the credentials presented by the requesting application.

The client application attaches credentials (or a reference to credentials) to the request message and digitally signs the message with the client's private key. When a service receives the message, it uses the public key, which is included with the X.509 certificate, to validate the signature. Additional validation may be required to ensure that the X.509 certificate has not expired and was issued by a CA that the service trusts.

Participants

Brokered authentication with X.509 certificates issued by a certificate authority in a PKI involves the following participants:

- **Certificate authority (CA)**. A CA is an authentication broker that is responsible for authenticating clients and issuing valid X.509 certificates.
- **Certificate store**. This is where the X.509 certificates are located.
- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.

Process

A mutually trusted CA must issue an X.509 certificate before brokered authentication using X.509 can complete. You can obtain an X.509 certificate in one of the following ways:

- Purchase an X.509 certificate from a public CA.
- Configure a PKI server, such as Windows Certificate Services, to create an X.509 certificate, and then use the PKI CA to sign the certificate.
- Use a tool such as MakeCert to create a self-signed certificate (this is not suitable for production purposes).

After an X.509 certificate is issued, local repositories, such as a machine certificate store, are used to store information about the X.509 certificate. The actual process of issuing and distributing X.509 certificates is beyond the scope of this pattern. For detailed information, see X.509 Technical Supplement in Chapter 7, "Technical Supplements."

The process of using an X.509 certificate for authentication is shown in Figure 1.10.

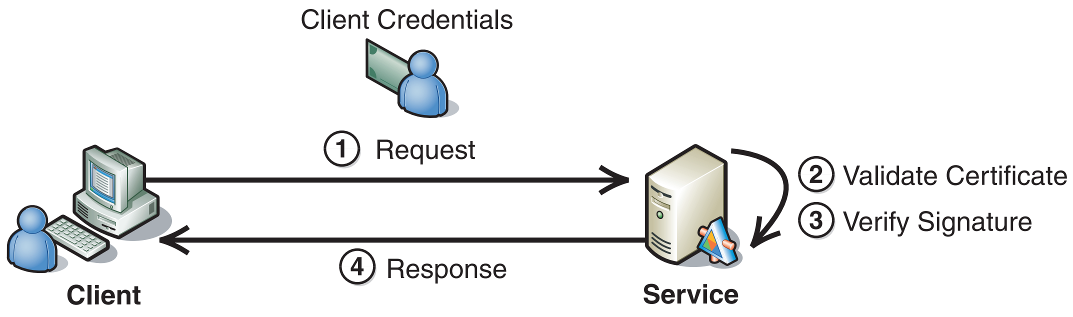


Figure 1.10
Authentication using an X.509 certificate

As illustrated in Figure 1.10, the following steps describe the process of authentication using an X.509 certificate:

1. The client sends a message to the service. The message includes the client's credentials, signed with the private key that is paired with the public key in the client's X.509 certificate. The client can also attach its X.509 certificate to the message if the service does not store or have access to the X.509 certificates out of band. If the X.509 certificate is not attached, the client attaches a certificate identifier to the request message so that the service can retrieve the client's X.509 certificate from a certificate repository and verify the message signature.
2. The service validates the certificate, by performing a number of checks, including:
   - Verifying that the certificate has not expired. If the expiration date in the certificate is past the current date, then the certificate is not valid.
   - Verifying that the certificate is internally consistent. The service checks that the data in the certificate has not been tampered with by verifying the certificate contents against the signature of the issuing CA.
   - Verifying the issuing CA of the client's X.509 certificate. This is done by comparing the issuer signature on the user's X.509 certificate with the X.509 certificate of the issuing CA. For this step to be of any value to either party, the CA that issued the client's X.509 certificate must be trusted by both the client and service.
   - Verifying that the issuing CA has not revoked the certificate. The service checks this by making sure that the X.509 certificate does not appear on a certificate revocation list (CRL) published by the issuing CA. The service can check the revocation status of the certificate by directly accessing it from the CA or by checking against a CRL that was previously downloaded from the issuing CA to the certificate repository used by the service to look up X.509 certificates.

3. The service uses the public key in the client's X.509 certificate to verify the client's signature. This allows the service to authenticate the client and ensure that the signed data has not been tampered with after the message was signed.
4. (Optional) The service may send a response back to the client.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Brokered Authentication: X.509 PKI pattern include the following:

- Authentication can occur over well known Internet firewall-friendly ports through well-known protocols (for example, HTTP/HTTPS over port 80/443).
- X.509 certificates can be used to authenticate clients and protect messages across organizational boundaries and security domains because the X.509 certificates are based on a broadly accepted standard. PKI using X.509 certificates has the capacity to establish a common basis of trust beyond the scope of individual organizations. Only a relatively small number of certificate issuers are widely trusted across public networks, which simplifies the management of trust with those issuers.
- The X.509 CA supports renewal and revocation of X.509 certificates, as follows:
  - An agent, acting on the client's behalf, can renew an X.509 certificate to extend the life time of the certificate. When an X.509 certificate is renewed, a new copy of the certificate is generated with a new expiration date, sometimes along with a corresponding new public/private key pair.
  - X.509 certificates may be revoked if any of the client's information in the X.509 certificate has changed or if the X.509 certificate's private key has been compromised.
- X.509 certificates can be distributed openly and used by anyone to encrypt messages to a client or to verify the digital signature of the client. For more information about protecting confidential data, see Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Digital signatures provide a means of supporting non-repudiation. This is because access to the private key is usually restricted to the owner of the key, which makes it easier to verify proof-of-ownership. For more information about non-repudiation, see Data Origin Authentication in Chapter 2, "Message Protection Patterns."
- Authentication does not require a direct relationship between every client and service.

Liabilities

The liabilities associated with the Brokered Authentication: X.509 PKI pattern include the following:

- Private keys need to be stored securely (such as on a smart card or your computer) and are therefore not as portable as passwords. An attacker could use a private key to impersonate the client. Therefore, you must make sure that the private key is not compromised.
- Generating and verifying digital signatures in X.509 is computationally intensive. If the client sends frequent request messages to the service during a normal interaction, you should consider a means to optimize communication between the two parties, such as secure conversation.
- Certificates by themselves are not well suited to provide role-based security, because role assignment tends to change relatively frequently and X.509 certificates typically have a long life time. However, you can supplement X.509 certificate authentication with a role store to provide more fine-grained authorization capabilities. One possible solution is to combine X.509 authentication with a Lightweight Directory Access Protocol (LDAP) directory or Active Directory with certificate mapping enabled.
- Organizations could require additional infrastructure to support an X.509 PKI. The benefits gained from using an X.509 PKI must be compared with the investment required to use it.

Security Considerations

Security considerations associated with the Brokered Authentication: X.509 PKI pattern include the following:

- It is critical to safeguard the private key associated with the X.509 certificate. If the private key is compromised, the integrity of the corresponding X.509 certificate is violated because another entity besides the client is capable of generating digital signatures that represent the client's identity. If a private key is compromised, the CA can revoke the X.509 certificate, which causes it to become unusable for encryption and digital signatures.
- The life time of an X.509 certificate is considerably greater than that of other authentication broker token types. Most tokens from an authentication broker expire minutes or hours from their time of issue, whereas an X.509 certificate can be valid for several months.
- Regardless of whether an X.509 certificate is renewed or revoked and a new X.509 certificate is re-issued, the X.509 certificate should use a newly generated public/private key pair. For existing X.509 certificates that are being renewed, this is known as re-keying the X.509 certificate.
- Only one copy of the client's X.509 certificates private key should exist when it is used to support non-repudiation through digital signatures. This private key should be accessible to the client only.

- If private keys are centrally managed — for example, by using a key escrow — and the centralized store is compromised, you may not be able to use digital signatures to strongly attribute an action to a specific party.
- In some cases, after a service has authenticated a client, it will need to authorize the client based on the client identity. The service must be able to either recognize the client individually or verify that the client belongs to a limited population. The service can accomplish this in one of the following ways:
  - By defining a policy that only allows requests to be processed that are signed by specific X.509 certificates.
  - By requiring verification of X.509 client certificates against a very restricted trust chain. This allows you to closely regulate the population of clients from which the server will accept requests. For more information about X.509 certificate trust chains and trust anchors, see X.509 Technical Supplement in Chapter 7, "Technical Supplements."
- Messages that are signed and encrypted with X.509 certificates are susceptible to *surreptitious forwarding* attacks. In this type of attack, the recipient of a signed and encrypted message decrypts the message, encrypts it using a third-party's public key, and then sends it on to that third party with the original signature still in the message. In this case, the message can appear as though it was sent to the third party from the original sender. To mitigate this type of attack, the original sender can sign some information that binds the message to the intended recipient, such as the WS-Addressing headers that specify the intended recipient of the message.
- If an authentication broker is compromised, the integrity of the trust that the broker provides is also compromised. If a CA is compromised, an attacker could issue certificates to himself/herself to act as a valid client within the CA's trust chain. An attacker could use these certificates to perform malicious actions while posing as a trusted client.
- You should use mutual authentication to be sure that each party using X.509 is who they claim to be. With mutual authentication, the client authenticates the service and the service authenticates the client. For authentication with X.509 certificates, each party must be able to verify a piece of signed data provided by the other party with that party's X.509 certificate. Alternatively, if only one party has an X.509 certificate, shared keys can be combined with X.509 certificates to provide mutual authentication. For an example of such an approach, see Implementing Message Layer Security with X.509 Certificates in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Related Patterns

Four types of patterns are related to this pattern: parent patterns, child patterns, alternate patterns, and patterns that use the Brokered Authentication: X.509 PKI pattern.

The following parent pattern is related to the Brokered Authentication: X.509 PKI pattern:

- **Brokered Authentication**. This pattern describes how to prove a client's identity to an authentication broker so that the broker can issue a security token.

The following child patterns are related to the Brokered Authentication: X.509 PKI pattern.

- **Implementing Message Layer Security with X.509 Certificates in WSE 3.0**. This pattern explains how to implement brokered authentication, authorization, data integrity, and data origin authentication using X.509 certificates in WSE 3.0.
- **Implementing Transport Layer Security Using X.509 Certificates and HTTPS**. This reference provides a concise reference on how to use SSL for data confidentiality and data integrity and how to use SSL client certificates for brokered authentication and data origin authentication.

The following alternate patterns are related to the Brokered Authentication: X.509 PKI pattern:

- **Brokered Authentication: Kerberos**. This pattern provides an alternative to X.509 based on the Kerberos authentication protocol.
- **Brokered Authentication: Security Token Service (STS)**. This pattern provides an alternative to X.509 that is highly interoperable between platforms, security protocols, and credential types.

The following pattern uses the Brokered Authentication: X.509 PKI pattern:

- **Implementing Direct Authentication with UsernameToken in WSE 3.0**. This pattern relies on relies on X.509 certificates, to ensure that sensitive credentials can be propagated securely.

Brokered Authentication: Security Token Service (STS)

Context

Web services need to authenticate clients in a heterogeneous environment so that additional controls such as authorization and auditing can be implemented. The organization has decided to use an *authentication broker* to provide a common access control infrastructure for a group of applications. The authentication broker negotiates trust between client applications and Web services; this removes the need for a direct relationship. The authentication broker should issue signed security tokens that can be used for authentication.

Problem

How does the Web service verify the credentials presented by the client?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Clients requiring authentication are implemented on a variety of platforms within the organization, and interoperability is required between those platforms**. Using a standards based mechanism for authentication helps ensure interoperability between different platforms.
- **The organization has identified a need for security tokens that are extensible and include claims that support additional security functions**. The authentication broker must be flexible enough to receive and issue tokens that support additional functionality such as authorization, auditing, and custom authentication.

The following condition is an additional reason to use the solution:

- **The environment includes organizational boundaries that are protected by firewalls**. The authentication broker must be able to issue security tokens that can traverse these boundaries, including passing through ports that are commonly enabled on firewalls.

The following conditions are not resolved by the base pattern, but they are resolved by the extensions provided at the end of this pattern:

- **Users access multiple clients that call Web services, resulting in the need for single sign on (SSO) capabilities**. To ensure a positive user experience, users should have to enter a user name and password only when logging on to a workstation; users should not have to re-enter them multiple times when accessing multiple clients.
- **The environment includes multiple security domains**. Clients must be able to obtain security tokens, so that resources such as services can be accessed in a different security domain using a security token issued by the authentication broker in its own security domain.

Solution

Use brokered authentication with a security token issued by a Security Token Service (STS). The STS is trusted by both the client and the Web service to provide interoperable security tokens.

The client sends an authentication request, with accompanying credentials, to the STS. The STS verifies the credentials presented by the client, and then in response, it issues a security token that provides proof that the client has authenticated with the STS. The client presents the security token to the Web service. The Web service verifies that the token was issued by a trusted STS, which proves that the client has successfully authenticated with the STS.

The protocol used for issuing security tokens is based on WS-Trust. WS-Trust is a Web service specification that builds on WS-Security. It describes a protocol used for issuance, exchange, and validation of security tokens. WS-Trust provides a solution for interoperability by defining a protocol for issuing and exchanging security tokens, based on token format, namespace, or trust boundaries.

In WS-Trust, the type of message sent to an STS to request issuance of a security token is known as a Request Security Token (RST) message. The RST message contains credentials for the client to be authenticated, such as the user ID and password contained in a UsernameToken. The response message from the STS is known as a Request Security Token Response (RSTR) message. The RSTR contains a security token, such as an XML Security Assertion Markup Language (SAML). For more information about WS-Trust, see *Web Services Trust Language (WS-Trust)* on MSDN.

SAML Tokens

SAML (Security Assertion Markup Language) tokens are standards-based XML tokens that are used to exchange security information, including attribute statements, authentication decision statements, and authorization decision statements. SAML tokens are also extensible; this means you can extend the schema of the token to meet additional requirements.

SAML tokens are important for Web service security because they provide cross-platform interoperability and a means of exchanging security information between clients and services that do not reside within a single security domain. They can be used as part of an SSO solution allowing a client to talk to services running on disparate technologies.

The SAML specifications cover a broad range of topics — from the format of the actual SAML token to a protocol that can be used for token request and issuance. Microsoft products use the WS-* specifications, which include the use of SAML assertions but not the SAML protocol. Instead of the SAML protocol, token issuance and federation uses the WS-Trust and WS-Federation set of specifications. Currently, ADFS in Windows Server 2003 R2 uses SAML 1.1 tokens and the WS-Federation passive client profile specification to enable SSO scenarios within Web applications. For more information about ADFS, see Introduction to ADFS on Microsoft TechNet. Future support for active client scenarios (such as SSO support for Web services) is under development.

(continued)

SAML Tokens *(continued)*

The SAML standard is still evolving from version to version, and the versions are not currently interoperable. At the time of this writing, there is increasing adoption of the SAML 1.1 specification, but implementations may need to be modified to support future versions of the SAML standard if SAML tokens are used.

For more information about the SAML 1.1 specification, including the protocol for request and issuance of SAML tokens, see the OASIS Web site.

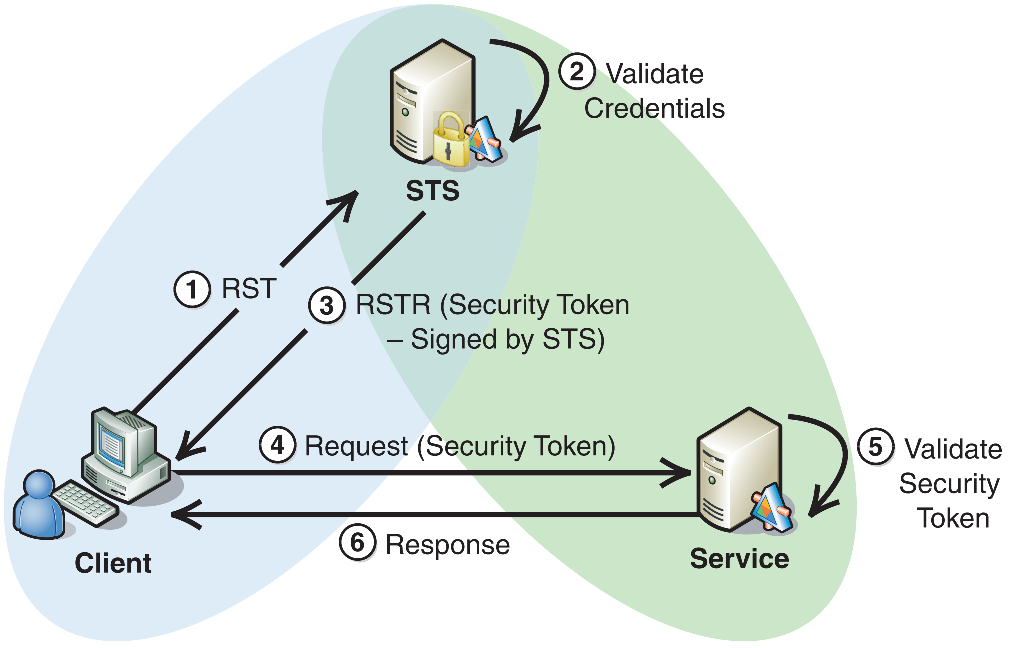
Participants

Brokered authentication with STS involves the following participants:

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **STS**. The STS is the Web service that authenticates clients by validating credentials that are presented by a client. The STS can issue to a client a security token for a successfully authenticated client.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.

Process

Figure 1.11 illustrates the process by which a security token is issued to the client by the STS and then used to authenticate with a service, which then returns a response to the client.



② Validate Credentials
STS
① RST
③ RSTR (Security Token – Signed by STS)
④ Request (Security Token)
⑤ Validate Security Token
Client
⑥ Response
Service

Figure 1.11
STS token issuance and request/response

As illustrated in Figure 1.11, the following steps describe STS token issuance and request/response process:

1. **The client initializes and sends authentication request to the STS**.
   The authentication request to the STS is in the form of an RST message. This step can be performed by presenting the client's identifier and proof-of-possession (such as user name and password) directly to the STS or by using a token issued by an authentication broker (such as an X.509 digital signature or Kerberos tokens).

   The RST message contains a security token that holds the client's credentials, which are required to authenticate the client. Claims in the client's credentials, such as a password, may be sensitive in nature, so it is very important to secure the RST. The specific security mechanism used for securing the RST depends on the relationship between the client and the STS. For example, the client and STS may use Kerberos tokens or X.509 certificates to sign and encrypt messages sent between them. For more information about securing messages, see Data Confidentiality and Data Origin Authentication in Chapter 2, "Message Protection Patterns."
2. **The STS validates the client's credentials**. After the STS determines that the client's credentials are valid, it may also decide whether to issue a security token for the authenticated client. For example, the STS may have a policy where it issues tokens only for users who belong to a specific role or for valid X.509 certificates that can be validated through a specific trust chain.
3. **The STS issues a security token to the client**. If the client's credentials are successfully validated, the STS issues a security token (such as a SAML token) in an RSTR message to the client; typically, the security token contains claims related to the client. The security token is usually signed by the STS; when the security token is signed by STS, the service can confirm that the token was issued by the STS and that the security token was not tampered with after it was issued.
4. **The client initializes and sends a request message to the service**. After the client receives a security token from the STS, it initializes a request message that includes the issued security token, and then it sends the request message to the service.
5. **The service validates the security token and processes the request**. The security token is validated by the service to verify that the token was issued by the trusted STS and that the token was not tampered with after it was issued. After the token is validated by the service, it is used to establish security context for the client, so the service can make authorization decisions or audit activity.
6. **(Optional) The service initializes and sends a response message to the client**.
   A response is not always required. Frequently, the response message contains sensitive data, so it should be secured.

A client may also specify the scope of the request for a security token to the STS. Scope is a value that identifies the target of the client; it can be as granular as a single operation of the Web service or as broad as an application domain. The token issued by the STS can contain usage constraints that correspond to the scope of the request.

Scope can be used to provide resource level authorization, with the STS comparing the value in the scope to a list of clients that are authorized to access the target.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Brokered Authentication: Security Token Service (STS) pattern include the following:

- This pattern provides a flexible solution for exchanging one type of security token for another to accomplish a variety of goals in a Web service environment, such as authentication, authorization, and exchanging session keys.
- The solution is not dependent on any one mechanism, such as the Kerberos protocol or X.509 to secure messages. This makes it easier to enable different authentication protocols to interoperate, by adding a level of abstraction on top of existing protocols.

Liabilities

The layer of abstraction provided by the STS means that the STS must use another underlying security protocol to provide functionality such as authentication and authorization. This can make the STS a more difficult solution to implement, particularly in cases where a custom solution is used.

Security Considerations

Security considerations associated with the Brokered Authentication: Security Token Service (STS) pattern include the following:

- Request and response messages between the client and the STS often contain sensitive information, such as user passwords and session encryption/signature keys, so they should be protected using data encryption and data origin authentication. For more information about data encryption, see Data Confidentiality in Chapter 2, “Message Protection Patterns.” For more information about data origin authentication, see Data Origin Authentication in Chapter 2, “Message Protection Patterns.”
- Request and response messages between the client and the STS and between the client and the service may also be susceptible to message replay attacks if communication is secured at the message layer. For information about preventing an attacker from replaying messages, see Message Replay Detection in Chapter 5, “Service Boundary Protection Patterns.”

Extensions

The extensions described here build on the base pattern to provide additional capabilities.

Extension 1 — Establishing a Secure Conversation

This extension can be used to establish a secure conversation with the STS. There are several reasons for establishing a secure conversation with the STS, including:

- Preventing the client from having to present a user name and password each time it accesses a different service. This could involve the client having to cache the client’s original credentials (which is not considered a safe security practice) or prompting users to provide their credentials each time.
- Improving performance when resource-intensive forms of credentials, such as X.509 digital signatures, are used. Creation and validation of X.509 digital signatures is a computationally intensive process, so performance can be improved if they are used less frequently.

In this extension, the client obtains a Security Context Token (SCT) (which demonstrates that the client has been authenticated) from the STS and caches it. After the client is authenticated with the STS, the client can use the session token to request a service token for communication with a service. The way the STS validates a security token presented by a client and issues service tokens is similar to how the Kerberos protocol validates a ticket-granting ticket and issues a service ticket. For more information about the Kerberos protocol, see Brokered Authentication: Kerberos in Chapter 1, "Authentication Patterns." Figure 1.12 illustrates this behavior.

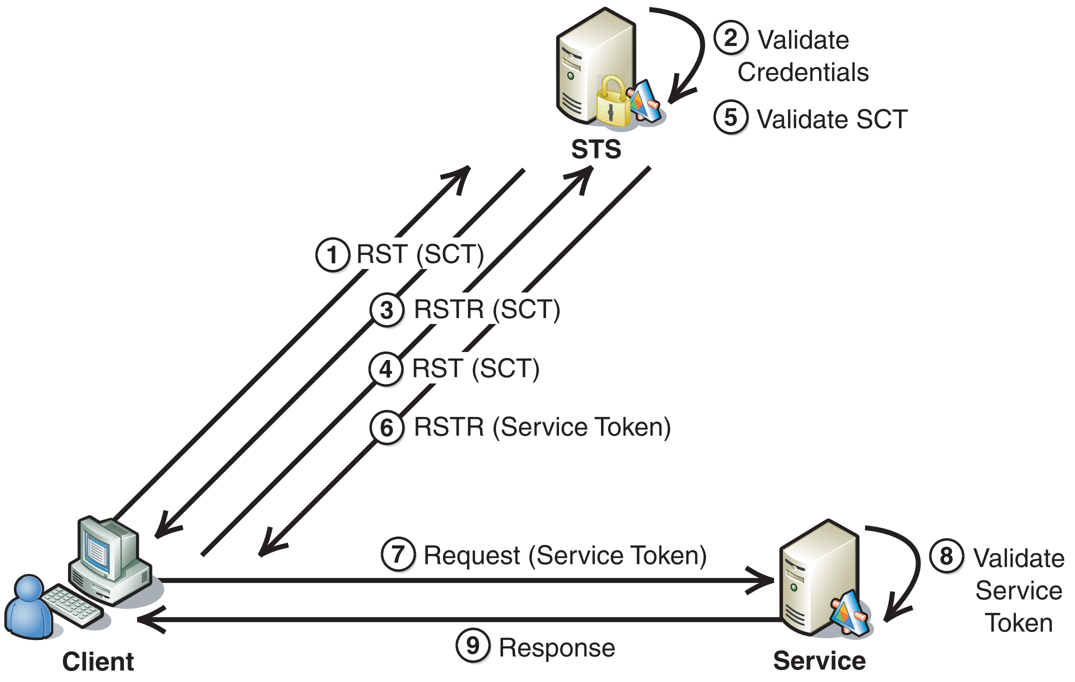


Figure 1.12
Establishing a secure conversation with the STS

This extension is based on the use of WS-SecureConversation to establish a session between the client and the service. WS-SecureConversation is a Web service specification that builds on WS-Security and WS-Trust. It describes how to establish a lightweight security context between two parties. The security context uses session keys; these session keys become the basis for encrypting and signing subsequent message exchanges, which results in more efficient secure communications between the two parties. For more information about WS-SecureConversation, see *Web Services Secure Conversation Language (WS-SecureConversation)* on MSDN.

Note: The security of any conversation depends on the key exchange mechanism. Typically, the key exchange mechanism is based on a key management infrastructure, such as one based on PKI or shared secrets.

The secure conversation extension can also be applied to the other direct authentication and brokered authentication patterns to optimize interactions between two parties. In this instance, the secure conversation is applied to demonstrate how a session can be established between a client and an STS.

While the establishment of a secure conversation with the STS logically includes three parties — the client, the STS and the service — the solution is typically implemented with the STS residing on the same node as the service. The STS issues SCTs to maintain state between the client and the STS, and it also issues service tokens for communication between the client and the service. The client can use the service token to authenticate with the service, and may even establish a secure conversation with the service.

This extension builds on the base pattern to provide additional capabilities. In addition to resolving the forces stated for the base pattern, it also resolves the following condition:

- **Users access multiple clients that call Web services, resulting in the need for single sign on (SSO) capabilities**. To ensure a positive user experience, users should have to enter a user name and password only when logging on to a workstation; users should not have to re-enter them multiple times when accessing multiple clients.

Process

This section describes the steps of the process illustrated in Figure 1.12. It demonstrates how the STS issues Security Context Tokens (SCTs) to allow the client to establish a session with STS. An SCT is a lightweight security token used to gain access to the STS and to optimize secure communications between the client and the STS.

Ad illustrated in Figure 1.12, the following steps describe the STS process:

1. **The client initializes and sends an authentication request to the STS**. The authentication request to the STS is in the form of an RST message. This step can be performed by presenting the client's identifier and proof-of-possession (such as user name and password) directly to the STS or by using a token issued by an authentication broker (such as an X.509 digital signature or Kerberos protocol Token). Information, such as a password, is sensitive in nature, so it is very important to secure the RST using message protection. For more information see Chapter 2, "Message Protection Patterns."
2. **The STS validates the client's credentials**. The client's credentials are a series of claims that prove the client's identity or confirm that the client has successfully authenticated with another trusted authentication broker such as an X.509 CA, a Kerberos KDC, or another STS.

3. **The STS issues a Security Context Token (SCT) to the client**. The SCT can be used by the client each time additional security tokens are required, instead of the client presenting the client's original credentials each time. The scope of the issued SCT is limited to the STS regardless of whether the client specified the scope in the initial RST. This prevents the client from using the SCT to directly access a service.
4. **The client caches the SCT**. By caching the SCT, the client can establish a session with the STS. Then the client can make subsequent requests to the STS without having to present the client's original credentials each time. The STS can include claims about the client in the SCT, which the STS can use to make authorization decisions.
5. **The client requests a service token from the STS to communicate with the service**. When the client attempts to communicate with a specific service, it sends another RST to the STS. This RST contains the SCT that was initially issued by the STS for the authenticated client. In the RST, the client specifies the target Web service as the scope of the request.
6. **The STS responds to the service token request**. The STS may have established a policy to determine whether the client is authorized to access the service that is specified in the scope. If the client is allowed access to that service, the STS issues to the client a security token that is used to authenticate with the service.
7. **The client initializes and sends request message to the service**. After the client obtains the required security token from the STS, it initializes a request message that includes the issued service token, and sends it to the service.
8. **The service validates service token and processes the request**. The service ensures that the security token was issued by the trusted STS and that the token was not tampered with after it was issued. After the token is validated by the service, it is used to establish a security context for the client, so the service can make authorization decisions or audit activity.
9. **Service initializes and sends response message to the client**. The client may not always expect a response from the service. The client knows whether to expect a response from the service if it has been specified in the Web service contract using Web Service Discovery Language (WSDL).

Note: It is also possible for the client to establish a secure conversation with the service. In this case, Step 7 would be preceded by a request for a SCT from the service using the newly issued service token (for example, a SAML token) as the basis for the initial authentication with the service.

Extension 2 — Web Service Federation

A client may need to communicate with Web services that operate across organizational boundaries. Typically, the different organizations each have their own autonomous security domains established. In this situation, the client is authenticated in the security domain where the client operates, but it must be authorized and audited within the security domain where the service operates for the client to be able to call the service.

This extension builds on the base pattern to provide additional capabilities. In addition to resolving the forces stated for the base pattern, it also resolves the following condition:

- **The environment includes multiple security domains**. Clients must be able to obtain security tokens, so that resources such as services can be accessed in a different security domain using a security token issued by the authentication broker in its own security domain.

With security federation, security claims can be propagated and consumed across different security domains to support identification, authentication, authorization, and auditing.

After the client is authenticated, the token obtained from the STS is exchanged for a token that is useable in the target security domain. Security domains can be federated in different ways, depending on the operating environment and the security requirements for applications within the federation.

Note: This extension demonstrates at a high level how a SAML STS can be used as part of a larger federation solution. As such, a comprehensive discussion of federation is outside the scope of this pattern. The federation solution described here would include support for additional capabilities, such as mapping role information from one domain into equivalent role information in another domain to provide support for authorization of the client.

Figure 1.13 illustrates an example of interaction between a client and a service in two different security domains that participate in a federation through their respective STSs:

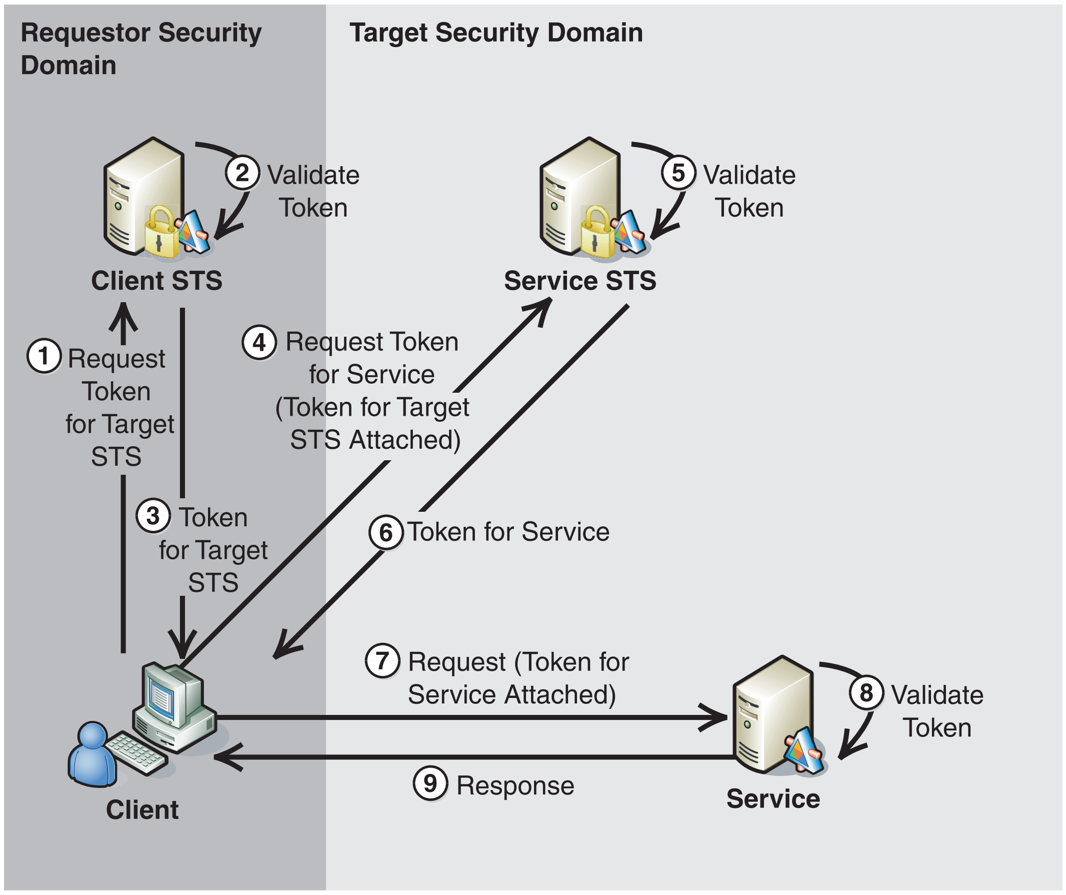


Figure 1.13
Obtaining a security token to authenticate with a service in a different security domain

Note: In this model of federation, the client is responsible for requesting the appropriate security token, which is consumable by the target Web service, as shown in Figure 1.13. This example can be used for active or passive clients. Support for passive clients is possible if the STS issues security tokens that are useable through HTTPS and can be cached by the browser.

As illustrated in Figure 1.13, the following steps describe Web service federation process:

1. **The client requests a security token to communicate with the STS in the target security domain**. The client presents authentication credentials, a security token previously issued by the STS, or a security token issued by another trusted authentication broker to obtain the security token for the target security domain.
2. **The STS in the client's domain validates the credentials or security token presented by the client**. The STS in the client's security domain may make authorization decisions about whether to issue a security token to the client for use in the security domain where the target service operates.
3. **The STS in the client's domain issues a security token to the client that is used to obtain a service token from the STS in the domain where the target service operates**. If the client is authenticated and authorized by the STS in the client's domain that STS issues a security token to the client to communicate with the STS in the target domain where the Web service operates. Based on the target security domain, the STS in the client's domain knows the type and scope of security token to issue.
4. **The client requests a security token from the STS in the target security domain**. The client presents the token issued by its STS to communicate with the STS of the target security domain.
5. **The target STS validates the client's security token**. The STS in the target security domain verifies that the token presented by the client originated from an STS in a trusted security domain. After the STS in the Web service's security domain validates the security token presented by the client, it may make an authorization decision about whether the client is authorized to access the requested service.
6. **The target STS issues a security token to communicate with the service**. If the target STS decides that the request is valid and the client is authorized to communicate with the service, it will issue a security token to the client that can be used to communicate with the service.
7. **The client sends a request message to the service**. The client attaches the security token it received from the STS in the target service's security domain to the request and sends it to the service.
8. **The service validates the security token attached to the request**. The service verifies that the token presented by the client was issued by a trusted STS.
9. **The service initializes and sends a response message to the client**.

Related Patterns

Three types of patterns are related to this pattern: parent patterns, child patterns, and alternate patterns.

The following parent pattern is related to the Brokered Authentication: Security Token Service (STS) pattern:

- **Brokered Authentication**. This pattern describes how to prove a client's identity to an authentication broker so that the broker can issue a security token.

The following child pattern is related to the Brokered Authentication: Security Token Service (STS) pattern:

- **Implementing Message Layer Security with a Security Token Service (STS) in WSE 3.0**. This pattern provides implementation guidelines for using an STS in WSE 3.0 to implement brokered authentication. This pattern is currently still in development, and is scheduled for completion in early 2006.

The following alternate patterns are related to the Brokered Authentication: Security Token Service (STS) pattern:

- **Brokered Authentication: Kerberos**. The Kerberos protocol provides an alternative to X.509 based on the Kerberos authentication protocol.
- **Brokered Authentication: X.509 PKI**. This pattern describes a specialized authentication broker based on the X.509 PKI standard.

More Information

For more information about authorization on the .NET Framework, see "Authentication and Authorization" in *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* on MSDN: *http://msdn.microsoft.com/practices/Topics/security/default.aspx?pull=/library/en-us/dnnetsec/html/SecNetch03.asp*.

For more information about Web services security, see OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/*.

For more information about the Kerberos protocol specifications, see RFC 1510: The Kerberos Network Authentication Service (V5): *http://www.faqs.org/rfcs/rfc1510.html*.

For more information about Kerberos authentication in Windows Server 2003, see "Kerberos Authentication Technical Reference" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/b748fb3f-dbf0-4b01-9b22-be14a8b4ae10.mspx*.

For a general overview of PKI technologies, see "PKI Technologies" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/6d5d9ef3-75ca-46c1-acf6-57dc7e9a6adf.mspx*.

For more information about WS-Trust, see *Web Services Trust Language (WS-Trust)* on MSDN: *http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-trust.pdf*.

For more information about ADFS, see "Introduction to ADFS" on Microsoft TechNet: *http://technet2.microsoft.com/WindowsServer/en/Library/c67c9b41-1017-420d-a50e-092696f40c171033.mspx*.

For more information about Security Assertion Markup Language (SAML), go to the OASIS Web site: *http://www.oasis-open.org/specs/index.php#samlv1.1*.

For more information about WS-SecureConversation, see *Web Services Secure Conversation Language (WS-SecureConversation)* on MSDN: *http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-secureconversation.pdf*.

For more information about SAML token profile 1.0, see *Web Security Services: SAML Token Profile* on the Oasis Web site: *http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf*.

2

Message Protection Patterns

Introduction

Web services send and receive plaintext messages over standard Internet protocols such as HTTP. Such plaintext messages can be intercepted by an attacker and potentially viewed or even modified for malicious purposes. By using message protection, you can protect sensitive data against threats such as eavesdropping and data tampering. This chapter provides design patterns for data confidentiality and data origin authentication.

This chapter provides design patterns for data confidentiality and data origin authentication. Figure 2.1 is a pattern map that illustrates these patterns.

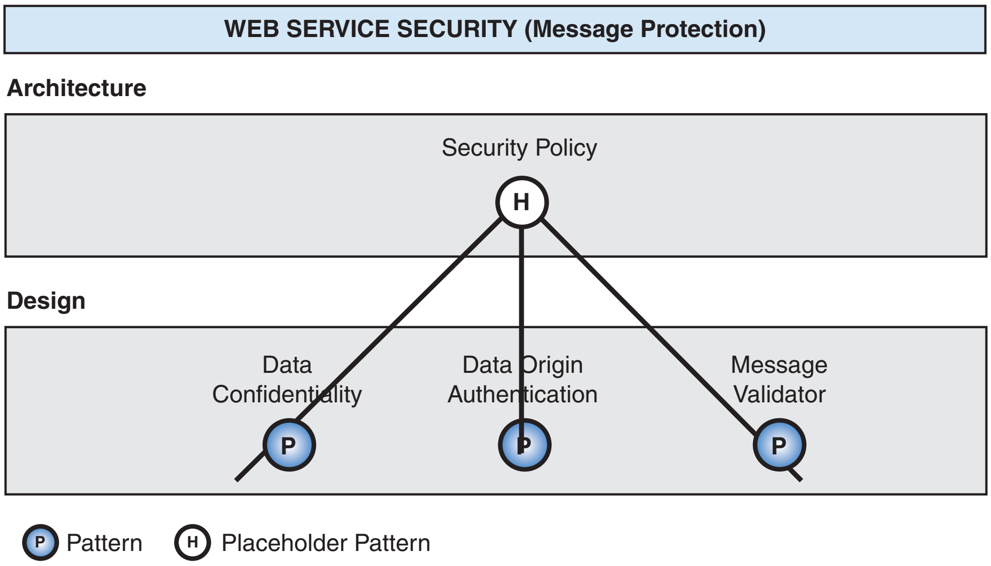


Figure 2.1
Message protection patterns

Note: The main factors that drive the type of message protection required are usually related to security policies within your organization and threat analysis performed for a particular application. It is strongly recommend that you perform a threat analysis to help understand your requirements. For more information about threat modeling, see Threat Modeling Web Applications on MSDN.

Data Integrity, Data Origin Authentication, and Data Confidentiality

Message protection can be divided into three main categories:

- Data integrity is the verification that a message has not changed in transit.
- Data origin authentication takes data integrity a step further and supports the ability to identify and validate the origin of a message.
- Data confidentiality is the encrypting of message data so that unauthorized entities cannot view the contents of the message.

As shown in Figure 2.1, a pattern does not exist for data integrity. Instead of creating a separate design pattern for data integrity, many of the implementation patterns in Chapter 3, "Implementing Transport and Message Layer Security," include data integrity as a step in the process. Because you should consider data integrity issues as you determine the message protection required in your environment, the decision matrix in Table 2.1 includes a Data Integrity column. The other two columns are mapped to design patterns in this chapter, which is consistent with other decision matrices.

Table 2.1 represents a decision matrix that lists security considerations related to message protection and how each one is supported by data integrity, data origin authentication, and data confidentiality.

Table 2.1: Message Protection Decision Matrix

Security Consideration	Data Integrity	Data Origin Authentication	Data Confidentiality
You want to verify that the contents of a message were not altered in transit.	Allows verification that a message has not changed in transit.	Supports the ability to verify that a message has not changed in transit and verify the origin of a message.	Encryption does not prevent the contents of a message from being altered.
You want to verify that the source of the data is from the sender you are authenticating and that the contents of a message were not altered in transit.	Allows verification that a message has not been changed, but this does not necessarily imply that the receiver can verify the source of the data.	Supports the ability to verify that a message has not changed in transit and verify the origin of a message.	Encryption does not prevent the contents of a message from being altered.

(continued)

Table 2.1: Message Protection Decision Matrix *(continued)*

Security Consideration	Data Integrity	Data Origin Authentication	Data Confidentiality
You want to restrict access to the contents of a message to authorized users only.	Does not provide the ability to protect message contents from unauthorized users.	Does not provide the ability to protect message contents from unauthorized users.	Confidentiality can be used to encrypt the contents of a message so that only authorized users can view the message contents.
You are implementing direct authentication using a shared secret and want to prevent an attacker getting the secret.	Generating signatures based on shared secrets that may have low entropy leaves the message vulnerable to offline cryptographic guessing attacks; as such, WSE 3.0 recommends you secure direct authentication using either HTTPS or **UsernameForCertificate** assertions.	Generating signatures based on shared secrets that may have low entropy leaves the message vulnerable to offline cryptographic guessing attacks; as such, WSE 3.0 recommends you secure direct authentication using either HTTPS or **UsernameForCertificate** assertions.	Encryption combined with data integrity and data origin authentication can be used to protect the shared secret.
You want to implement the Message Replay Protection pattern to prevent an attacker from maliciously replaying messages. Replay detection depends on the ability to uniquely identify messages.	This option is often implemented using a hashing function that provides a unique identifier that can be used to determine if the same message is received multiple times.	This option is often implemented using a hashing function or digital signature that provides a unique identifier that can be used to determine if the same message is received multiple times.	Not applicable.

From the decision matrix in Table 2.1, you can see that data confidentiality is recommended during authentication and any time sensitive data is sent in a message.

The remainder of this chapter focuses on the following design patterns:

- Data Confidentiality
- Data Origin Authentication

Data Confidentiality

Context

Data passes between a client and a Web service, sometimes through one or more intermediaries. Messages may also be kept in repositories, such as message queues or databases. Some of the data within the messages is considered to be sensitive in nature. There is a risk that an attacker can gain access to sensitive data, either by eavesdropping on the network or accessing a repository.

Problem

How do you protect data within a message from being disclosed to unintended parties?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Disclosure of sensitive data can result in loss or damage, such as identity theft, lawsuits, loss of business, or regulatory fines**. Any data that contains sensitive information must be protected from unauthorized users.
- **Sensitive data may pass across the network**. Sensitive data must be protected from disclosure in transit. An eavesdropper can gain access to sensitive data whenever it leaves a secure area (such as a protected memory space) or crosses a non-secure communication line (such as a public network).
- **Sensitive data may be persisted for short periods of time, such as in a message queue, or over longer periods of time in a database or a file**. Sensitive data must be protected from disclosure in locations where it is persisted.

Solution

Use encryption to protect sensitive data that is contained in a message. Unencrypted data, which is known as plaintext, is converted to encrypted data, which is known as ciphertext. Data is encrypted with an algorithm and a cryptographic key. Ciphertext is then converted back to plaintext at its destination.

Participants

Data confidentiality involves the following participants:

- **Sender**. The sender is the originator of a message. A client can send a request message to a Web service, and a Web service can send a response message back to a client that has sent a request message.
- **Recipient**. The recipient is the entity that receives a message from the sender. A Web service is the recipient of a request message that is sent by a client, and a client is the recipient of a response message that it receives from a Web service.

Process

You can apply data confidentiality in two steps:

1. **Encrypting the data**. In this step, the sender converts plaintext to ciphertext, rendering it unintelligible to parties other than the intended recipient.
2. **Decrypting the data**. In this step, ciphertext is rendered intelligible to the intended recipient by converting it back to plaintext.

You can use two types of cryptography to provide data confidentiality: symmetric and asymmetric. While both symmetric cryptography and asymmetric cryptography follow the same basic process, they each have their own unique characteristics.

Symmetric Cryptography

With symmetric cryptography, both the sender and recipient share a key that is used to perform both encryption and decryption. Symmetric cryptography is commonly used to perform encryption. It also provides data integrity when symmetric keys are used in conjunction with other algorithms to create Message Authentication Codes (MACs). For more information about MACs, see Data Origin Authentication in Chapter 2, "Message Protection Patterns."

Figure 2.2 illustrates the process of encrypting and decrypting data with a shared secret key.

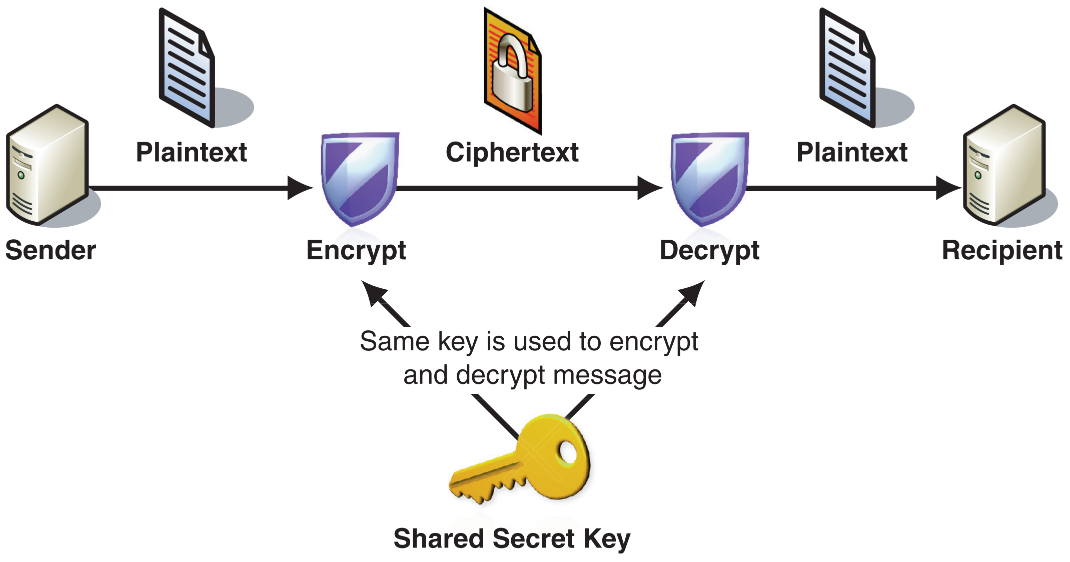


Figure 2.2
The process of symmetric encryption

As illustrated in Figure 2.2, symmetric encryption involves the following steps:

1. The sender creates a ciphertext message by encrypting the plaintext message with a symmetric encryption algorithm and a shared key.
2. The sender sends the ciphertext message to the recipient.
3. The recipient decrypts the ciphertext message back into plaintext with a shared key.

Numerous symmetric algorithms are currently in use. Some of the more common algorithms include Rijndael (AES) and Triple DES (3DES). These algorithms are designed to perform efficiently on common hardware architectures.

Symmetric cryptography is comparatively simple in nature, because the secret key that is used for both encryption and decryption is shared between the sender and the recipient. However, before communication can occur, the sender and the recipient must exchange a shared secret key. In some cases (such as SSL), asymmetric cryptography can be used to ensure that the initial key exchange occurs over a secure channel.

Asymmetric Cryptography

With asymmetric cryptography (also known as public key cryptography), the sender encrypts data with one key, and the recipient uses a different key to decrypt ciphertext. The encryption key and its matching decryption key are often referred to as a public/private key pair.

Note: In addition to providing encryption, you can use public key cryptography to provide digital signatures, facilitating nonrepudiation, and for key management purposes. For more information, see Data Origin Authentication in Chapter 2, "Message Protection Patterns."

The public key of the recipient is used to encrypt data. It can be openly distributed to those who want to encrypt a message to the recipient. The private key of the recipient is used to decrypt messages, and only the recipient must be able to access it.

Figure 2.3 illustrates the process of asymmetric encryption and asymmetric decryption.

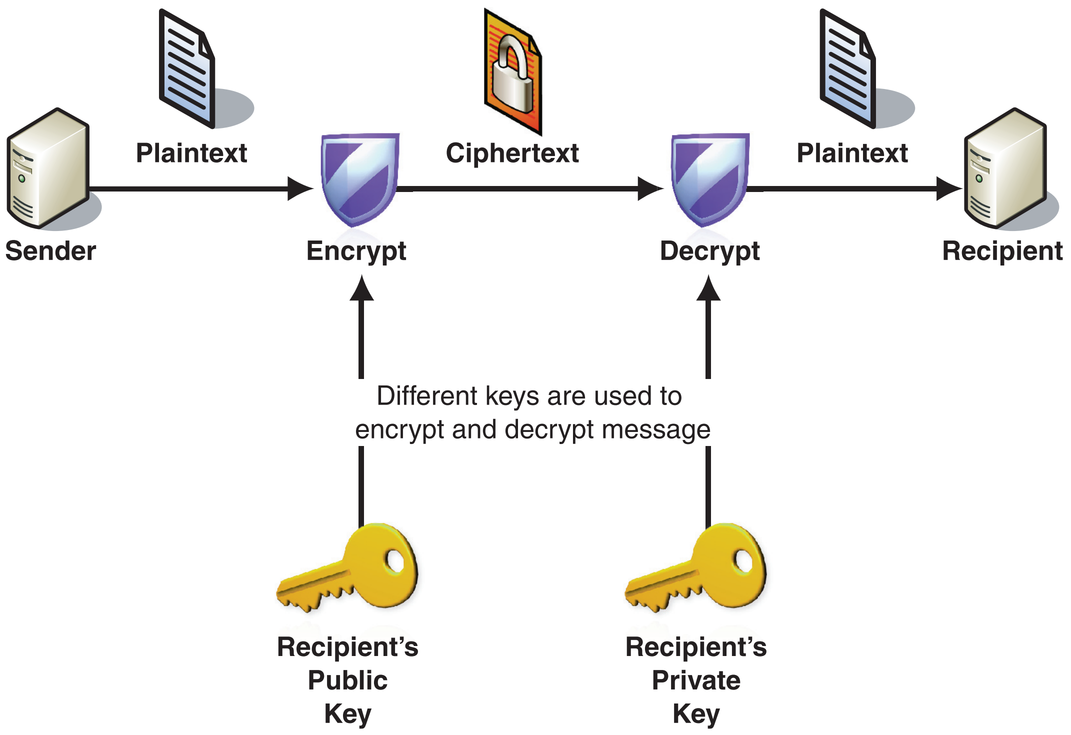


Figure 2.3
The process of asymmetric encryption

As illustrated in Figure 2.3, asymmetric encryption involves the following steps:

1. The sender creates a ciphertext message by encrypting the plaintext message with an asymmetric encryption algorithm and the recipient's public key.
2. The sender sends the ciphertext message to recipient.
3. The recipient decrypts the ciphertext message back to plaintext using the private key that corresponds to the public key that was used to encrypt the message.

Few asymmetric algorithms are currently in use. The most commonly used asymmetric algorithm is the RSA algorithm.

Asymmetric encryption requires more processing resources than symmetric encryption. For this reason, asymmetric encryption is usually optimized by adding a one time high-entropy symmetric key to encrypt a message and then asymmetrically encrypting the shared key. This reduces the size of the data that is asymmetrically encrypted and also improves performance.

In cases where more than one message exchange occurs between two parties, a high-entropy shared secret can be negotiated between a sender and a receiver. In this case, the first exchange includes a shared secret that is encrypted asymmetrically and based on the shared secret, additional message exchanges are performed symmetrically. Key derivation techniques are often used to add variability to shared secrets that are used over multiple message exchanges. For more information, see "Extension 1 — Establishing a Secure Conversation" in Brokered Authentication: Security Token Service (STS) in Chapter 1, "Authentication Patterns."

Example

Global Bank publishes a Web service to provide business customers with the ability to upload payroll account transfers. Direct deposit account information is considered very sensitive for both the business and the customer. Compromising this information can result in unauthorized account activity or disclosure of employee salary information. For this reason, Global Bank requires that any messages containing account data are encrypted as they pass between clients and the Web service to provide data confidentiality.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

By blocking unauthorized parties from viewing messages, you can prevent financial loss and legal liability because of the disclosure of sensitive information.

Liabilities

The liabilities associated with the Data Confidentiality pattern include the following:

- Cryptography operations are computationally intensive and impact system resource usage. This affects the scalability and performance of the application.
- Key management, which safeguards encryption keys from being compromised, can have significant administrative overhead. Factors that affect the administrative complexity of key management include:
  - The number and type of keys used.
  - The type of encryption used (symmetric or asymmetric).
  - The key management infrastructure in use.

Security Considerations

Security considerations associated with the Data Confidentiality pattern include the following:

- Encryption does not prevent data tampering. For example, a man-in-the-middle attack can replace the bits in transit, which can cause the receiver to decrypt the data to something other than the original plaintext. Without data origin authentication protection, the receiver has no way to verify that decrypted ciphertext is the same as the original plaintext. For this reason, the implementation patterns that implement data confidentiality also implement data origin authentication.
- If too much data is encrypted with the same symmetric key, an attacker can intercept several messages and attempt to cryptographically attack the encrypted messages, with the goal of obtaining the symmetric key. To minimize the risk of this type of attack, you should consider generating session-based encryption keys with a relatively short life span. Typically, these session keys are derived from a master symmetric key, such as a shared identity secret. Usually, the session key is exchanged by using asymmetric encryption during the initial interaction of a sender and recipient. Session keys should be discarded and replaced at regular intervals, based on the amount of data or number of messages that they are used to encrypt.
- Much of the strength of symmetric encryption algorithms comes from the randomness of their encryption keys. If keys originate from a source that is not sufficiently random, attackers may narrow down the number of possible values for the encryption key. This can make it possible for a brute force attack to discover the key value from encrypted messages that the attacker has intercepted. For example, a user password that is used as an encryption key can be very easy to attack because user passwords are typically a non-random value of relatively small size that a user can remember without writing it somewhere.
- You should use published, well-known encryption algorithms that have withstood years of rigorous attacks and scrutiny. Use of encryption algorithms that have not been subjected to rigorous review by trained cryptologists may contain undiscovered flaws that are easily exploited by an attacker.
- Each country may recognize different standards for data privacy/protection. For example, in the U.S., regulations such as Sarbanes-Oxley, HIPAA, and the Privacy Act of 1974 require that measures are taken to prevent disclosure of sensitive personal information or that there is accountability for the management of sensitive data. In the European Union (EU), regulations such as the Data Protection Directive enforce stringent standards for data privacy.

Related Patterns

The following child patterns are related to the Data Confidentiality pattern:

- **Implementing Direct Authentication with UsernameToken in WSE 3.0**. This pattern focuses on using direct authentication at the message layer in WSE 3.0.
- **Implementing Message Layer Security with X.509 Certificates in WSE 3.0**. This pattern provides guidelines for implementing brokered authentication, authorization, data integrity, and data origin authentication with X.509 certificates in WSE 3.0.
- **Implementing Message Layer Security with Kerberos in WSE 3.0**. This pattern provides guidelines for implementing brokered authentication, authorization, data integrity, and data origin authentication with the Kerberos version 5 protocol in WSE 3.0.
- **Implementing Transport Layer Data Confidentiality Using HTTPS**. This reference provides concise information about using data confidentiality and integrity with HTTPS.
- **Implementing Transport Layer Security Using X.509 Certificates and HTTPS**. This reference provides concise information about how to use SSL for data confidentiality and data integrity. It includes information about how to use SSL client certificates for brokered authentication and data origin authentication.

Data Origin Authentication

Context

Data passes between a client and a Web service, sometimes through one or more intermediaries. The data contained in the request message from the client influences the Web service's behavior. There is a risk that an attacker could manipulate messages in transit between the client and the Web service to maliciously alter the behavior of the Web service. Message manipulation can take the form of data modification within the message, or even substitution of credentials, to change the apparent source of the request message.

Problem

How do you prevent an attacker from manipulating messages in transit between a client and a Web service?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **An altered message can cause the message recipient to behave in an unintended and undesired way**. The message recipient should verify that the incoming message has not been tampered with.
- **An attacker could pose as a legitimate sender and send falsified messages**. The message recipient should verify that incoming messages originated from a legitimate sender.

The following condition is an additional reason to use the solution:

- **The organization may need to trace particular actions to a specific client or service**. A record of transactions allows an organization to provide evidence that a particular action was requested and/or performed. This could be useful if a user denies that he or she performed an action or if a client needs to verify that a service has performed a specific task.

Solution

Use data origin authentication, which enables the recipient to verify that messages have not been tampered with in transit (data integrity) and that they originate from the expected sender (authenticity).

In cases where the client denies having performed the action (nonrepudiation), you can use digital signatures to provide evidence that a client has performed a particular action that is related to data. Digital signatures can be used for nonrepudiation purposes, but they may not be sufficient to provide legal proof of nonrepudiation. By itself, a digital signature is just a mechanism to capture a client's association to data. In cases where data has been digitally signed, the degree to which an individual or organization can be held accountable is established in an agreement between the party that requires digital signatures and the owner of the digital signature.

Security Concepts

Proof-of-possession is a value that a client presents to demonstrate knowledge of either a shared secret or a private key to support client authentication.

Proof-of-possession using a shared secret can be established using the actual shared secret, such as a user's password, or a password equivalent, such as a digest of the shared secret, which is typically created with a hash of the shared secret and a salt value.

Proof-of-possession can also be established using the XML signature within a SOAP message where the XML signature is generated symmetrically based on the shared secret, or asymmetrically based on the sender's private key.

Participants

Data origin authentication involves the following participants:

- **Sender**. The sender is the originator of a message. A client can send a request message to a Web service, and a Web service can send a response message back to the client that has sent the request message.
- **Recipient**. The recipient is the entity that receives a message from the sender. A Web service is the recipient of a request message sent by a client. A client is the recipient of a response message that it receives from a Web service.

Process

Two types of signatures can be used to sign a message: symmetric and asymmetric.

Note: The following discussion refers to both XML signatures and digital signatures. XML signatures are used for SOAP message security with either a symmetric algorithm or an asymmetric algorithm. Digital signatures are created explicitly with an asymmetric algorithm and may or may not be used for SOAP message security.

Symmetric Signatures

A symmetric signature is created by using a shared secret to sign and verify the message. A symmetric signature is commonly known as a Message Authentication Code (MAC). A MAC is created by computing a checksum with the message content and the shared secret. A MAC can be verified only by a party that has both the shared secret and the original message content that was used to create the MAC.

The most common type of MAC is a Hashed Message Authentication Code (HMAC). The HMAC protocol uses a shared secret and a hashing algorithm (such as MD5, SHA-1, or SHA-256) to create the signature, which is added to the message. The message recipient uses the shared secret and the message content to verify the signature by recreating the HMAC and comparing it to the HMAC that was sent in the message.

If security is your primary consideration for choosing a hashing algorithm for an HMAC, you should use SHA-256 where possible for the hashing algorithm to create an HMAC. This is because it is the least likely algorithm to produce collisions (when two different pieces of data produce the same hash value). MD5 provides a high-performance method for creating checksums, though it is not a good choice for use as an HMAC because it can be compromised by brute force attack in a relatively short period of time. SHA-1 is currently the most widely adopted algorithm, so it may be required for interoperability reasons. Because of recent advances in cryptographic attacks against SHA-1, there is movement toward adopting more secure hash algorithms, such as SHA-256, as the recommended standard.

To protect a signature from offline cryptanalysis — especially those created with an older hash algorithm such as MD5 or SHA1 — the hash value should be encrypted as sensitive data. The shared key and algorithm that are used to encrypt the hash may depend on the symmetric algorithm used to encrypt sensitive data. (For more information, see Data Confidentiality in Chapter 2, "Message Protection Patterns.") When it is used to create an HMAC, the names of these algorithms are preceded by the term "HMAC" (for example, HMAC SHA-1 or HMAC MD5).

Figure 2.4 illustrates the process of using a MAC to sign a message.

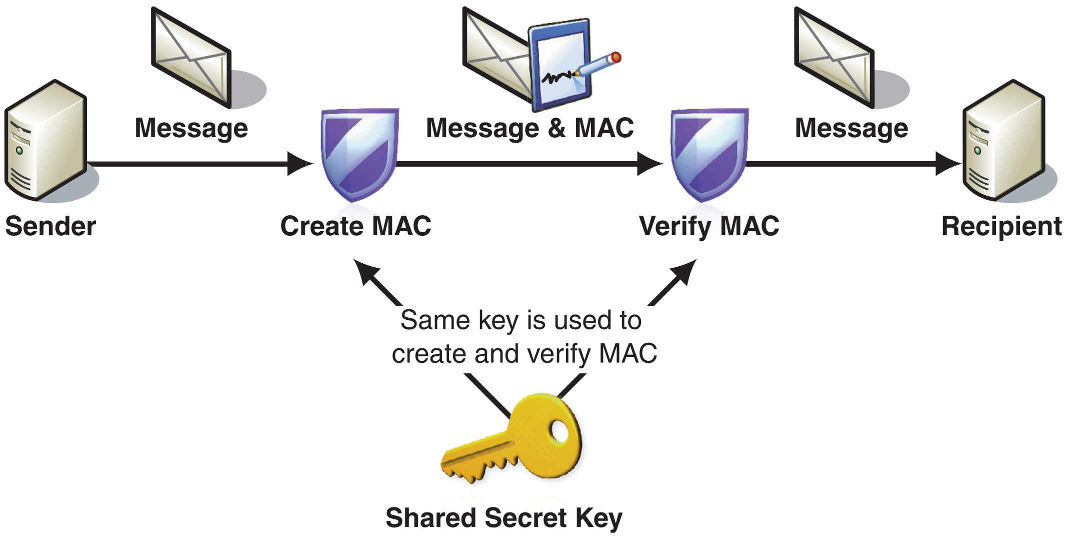


Figure 2.4
Signing a message using a symmetric signature

As illustrated in Figure 2.4, signing a message using a symmetric signature involves the following steps:

1. The sender creates a MAC using a shared secret key and attaches it to the message.
2. The sender sends the message and MAC to the recipient.
3. The recipient verifies that the MAC that was sent with the message by using the same shared secret key that was used to create the MAC.

By signing with a shared secret, both data integrity and data origin authenticity are provided for the signed message content. However, symmetric signatures are not usually used to provide nonrepudiation because shared secrets are known by multiple parties. This makes it more difficult to prove that a specific party used the shared secret to sign the message.

Asymmetric Signatures

An asymmetric signature is processed with two different keys; one key is used to create the signature and the other key is used to verify the signature. The two keys are related to one another and are commonly referred to as a public/private key pair. The public key is generally available and can be distributed with the message; the private key is kept secret by the owner and is never sent in a message. A signature that is created and verified with an asymmetric public/private key pair is referred to as a digital signature.

Figure 2.5 illustrates the process of using asymmetric keys to sign a message.

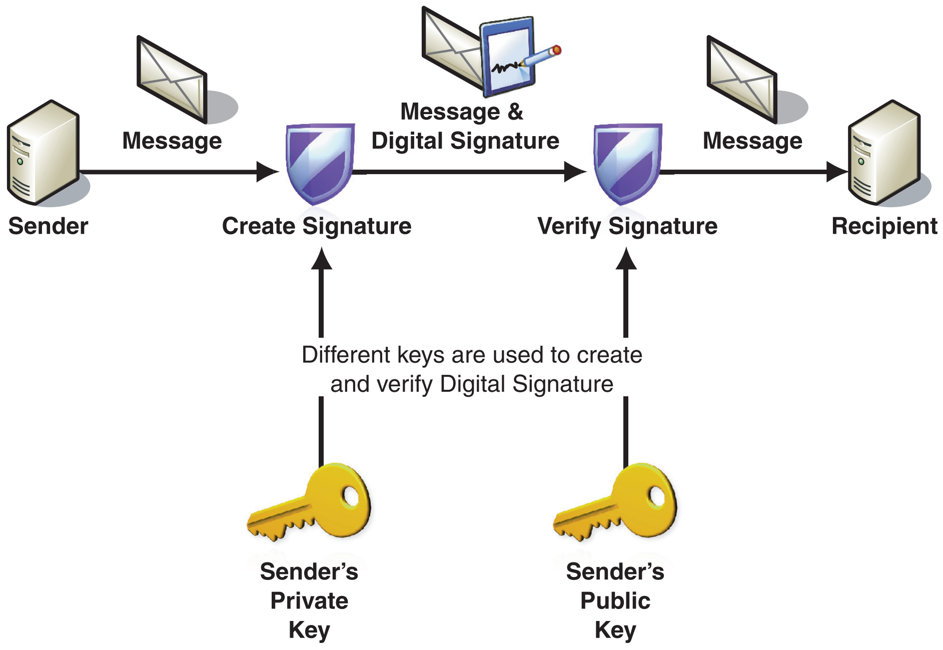


Figure 2.5
Signing a message with an asymmetric signature

As illustrated in Figure 2.5, signing a message with an asymmetric signature involves the following steps:

1. The sender signs the message content using the sender's private key and attaches it to the message.
2. The sender sends the message and digital signature to the recipient.
3. The recipient verifies the digital signature using the sender's public key that corresponds to the private key that was used to sign the message.

The algorithm that is most commonly used to create a digital signature is the Digital Signature Algorithm (DSA). DSA uses the public/private key pairs created for use with the RSA algorithm to create and verify signatures. For more information, see Data Confidentiality in Chapter 2, "Message Protection Patterns."

For both signing and encryption purposes, asymmetric keys are often managed through a Public Key Infrastructure (PKI). Information that describes the client is bound to its public key through endorsement from a trusted party to form a certificate. Certificates allow a message recipient to verify the private key in a client's signature using the public key in the client's certificate. For more information about X.509, see X.509 Technical Supplement in Chapter 7, "Technical Supplements."

Typically, digital signatures are used to support requirements for nonrepudiation. This is because access to the private key is usually restricted to the owner of the key, which makes it easier to verify proof-of-ownership.

Asymmetric signatures require more processing resources than symmetric signatures. For this reason, asymmetric signatures are usually optimized by hashing the message content and then asymmetrically signing the hash. This reduces the size of the data that the asymmetric operation is applied to.

In cases where more than one message is exchanged, it is also possible to first exchange a high-entropy shared secret that is encrypted asymmetrically. Based on the shared secret, additional message exchanges are secured symmetrically. Key derivation techniques are often used to add variability to shared secrets that are used over multiple message exchanges. For an example of this case, see "Extension 1 — Establishing a Secure Conversation" in Brokered Authentication: Security Token Service (STS) in Chapter 1, "Authentication Patterns." It is important to remember that this type of optimization can remove the ability of asymmetric signatures to isolate which of the two parties signed a message.

Example

When using message layer authentication, it is often necessary to include Data Origin authentication as part of the authentication process. One example of this is the use of X.509 certificates to perform message layer authentication. X.509 is based on public key cryptography, so the type of data origin authentication that is used is an asymmetric signature.

For example, a business customer at a bank may sign payroll transfers using his or her certificate private key. The bank can then verify that the payroll transfer request came from the correct business customer and that the message had not been tampered with in transit between the business customer and the bank.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The Data Origin Authentication pattern makes it possible for the recipient to detect whether a message has been tampered with. Also, the origin of the message can be traced to an identifiable source.

Liabilities

The liabilities associated with the Data Origin Authentication pattern include the following:

- Cryptographic operations, such as data signing and verification, are computationally intensive processes that impact system resource usage. This affects the scalability and performance of the application.
- Key management, which is responsible for maintaining the integrity of keys, can have a significant administrative overhead. Factors that affect the administrative complexity of key management include:
  - The number and type of keys used.
  - The type of cryptography used (symmetric or asymmetric).
  - The key management infrastructure in use.

Security Considerations

Security considerations associated with the Data Origin Authentication pattern include the following:

- If a message is being signed, you should ensure that the signature within the message is encrypted. In many cases, a signature that is not encrypted can be the target of a cryptographic attack.
- If too much data is encrypted with the same symmetric key, an attacker can intercept several messages and attempt to cryptographically attack the encrypted messages, with the goal of obtaining the symmetric key. To minimize the risk of this type of attack, you should consider generating session-based encryption keys that have a relatively short life span. Typically, these session keys are derived from a master symmetric key such as a shared identity secret. Usually, the session key is exchanged using asymmetric encryption during the initial interaction between a sender and recipient. Session keys should be discarded and replaced at regular intervals, based on the amount of data or the number of messages that they are used to encrypt.
- Much of the strength of symmetric encryption algorithms comes from the randomness of their encryption keys. If keys originate from a source that is not sufficiently random, attackers may narrow down the number of possible values for the encryption key. This makes it possible for a brute force attack to discover the key value of encrypted messages that the attacker has intercepted. For example, a user password that is used as an encryption key can be very easy to attack because user passwords are typically a non-random value of relatively small size that a user can remember it without writing it somewhere.
- You should use published, well-known encryption algorithms that have withstood years of rigorous attacks and scrutiny. Use of encryption algorithms that have not been subjected to rigorous review by trained cryptologists may contain undiscovered flaws that are easily exploited by an attacker.

Related Patterns

The following child patterns are related to the Data Origin Authentication pattern:

- **Implementing Direct Authentication with UsernameToken in WSE 3.0**. This pattern focuses on using direct authentication to verify message signatures at the message layer in WSE 3.0.
- **Implementing Message Layer Security with Kerberos in WSE 3.0**. This pattern provides guidelines for implementing brokered authentication, authorization, data integrity, and data origin authentication with the Kerberos version 5 protocol in WSE 3.0.

More Information

For more information about threat modeling, see "Threat Modeling Web Applications" on MSDN: *http://msdn.microsoft.com/practices/Topics/security/default.aspx?pull=/library/en-us/dnpag2/html/tmwa.asp*.

For more information about WS-Security version 1.0, see the OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/specs/index.php#wssv1.0*.

For more information about threats and countermeasures, see Chapter 2, "Threats and Countermeasures," of *Improving Web Application Security: Threats and Countermeasures* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh02.asp*.

For more information about HMAC, see RFC 2104 — HMAC: Keyed Hashing for Message Authentication: *http://www.ietf.org/rfc/rfc2104.txt?number=2104*.

For more information about WS-Security version 1.0, see the OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/specs/index.php#wssv1.0*.

For more information about threats and countermeasures, see the following:

- *Security Challenges, Threats and Countermeasures Version 1.0* on the WS-I Web site: *http://www.ws-i.org/Profiles/BasicSecurity/SecurityChallenges-1.0.pdf*.
- Chapter 2, "Threats and Countermeasures," of *Improving Web Application Security: Threats and Countermeasures* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh02.asp*.

3

Implementing Transport and Message Layer Security

Introduction

This chapter builds on the first two chapters in this guide to demonstrate how you can implement the patterns described in those chapters using Microsoft technologies. The chapter is divided into two sections. The first section provides three comprehensive composite patterns that focus predominantly on message layer security. These composite patterns are implementations of a number of different design patterns. The second section, “References for Transport Layer Security” discusses how you can solve many of the same security challenges using transport layer security.

Figure 3.1 is a pattern map that illustrates the composite patterns and references related to direct authentication.

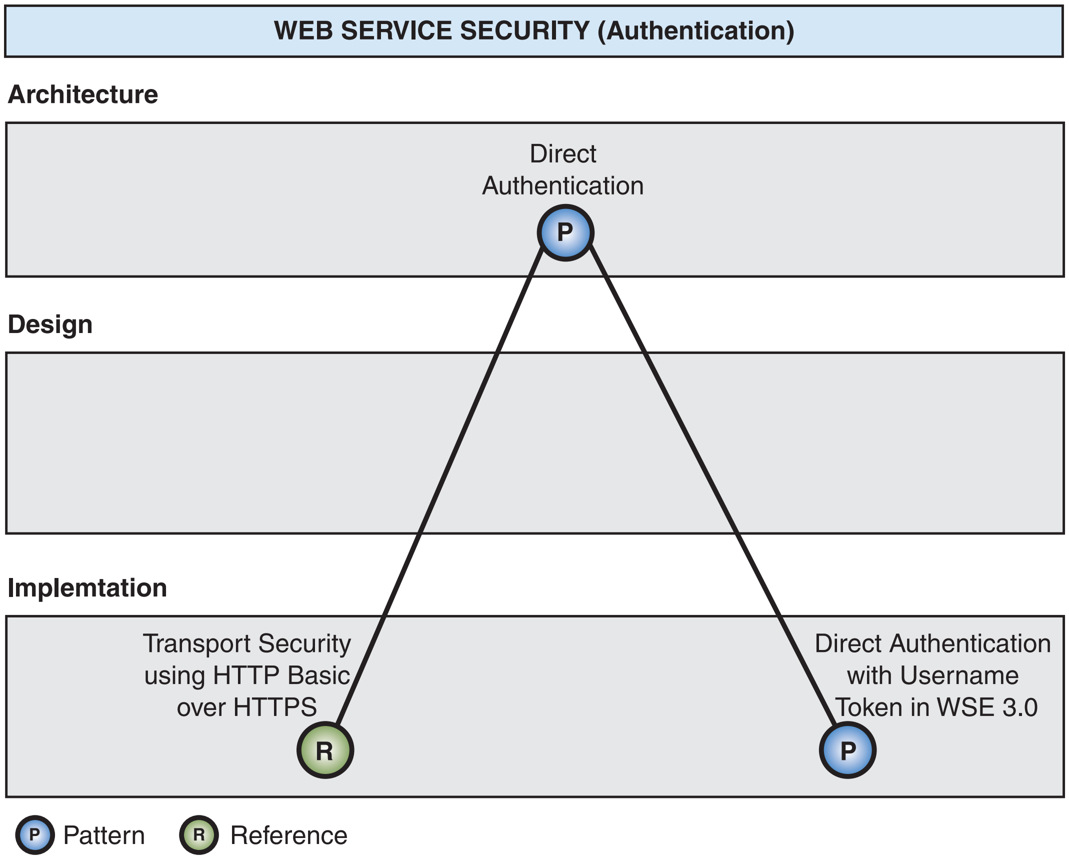

WEB SERVICE SECURITY (Authentication)
Architecture
Direct Authentication
P
Design
Implemtation
Transport Security using HTTP Basic over HTTPS
R
Direct Authentication with Username Token in WSE 3.0
P
P Pattern
R Reference

Figure 3.1
Direct authentication patterns and references

Figure 3.2 is a pattern map that illustrates the design patterns, composite patterns, and references related to brokered authentication.

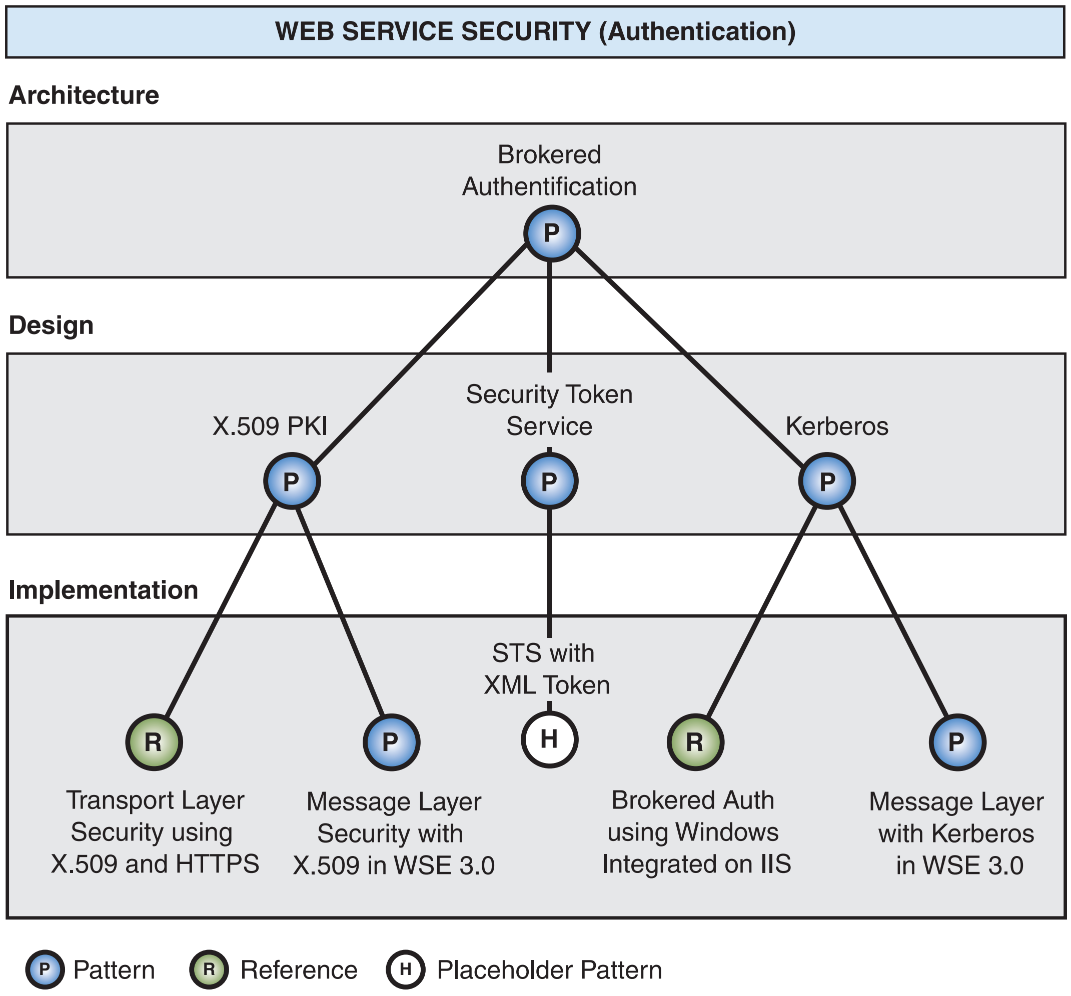

WEB SERVICE SECURITY (Authentication)
Architecture
Brokered Authentification
P
Design
X.509 PKI
P
Security Token Service
P
Kerberos
P
Implementation
R
Transport Layer Security using X.509 and HTTPS
P
Message Layer Security with X.509 in WSE 3.0
STS with XML Token
H
R
Brokered Auth using Windows Integrated on IIS
P
Message Layer with Kerberos in WSE 3.0
P Pattern
R Reference
H Placeholder Pattern

Figure 3.2
Brokered authentication patterns and references

Note: An implementation pattern for Security Token Service (STS) is due for release early in 2006.

Important Concepts

To fully understand transport layer security versus message layer security, it is important to understand the following concepts:

- **Credentials**. Credentials are a set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential.
- **Digital signature**. This is an asymmetric signature that is created with the private key of a client. Digital signatures can be used to support nonrepudiation requirements.
- **Security token**. A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential. Most security tokens will also contain additional information that is specific to the authentication broker that issued the token.
- **Protection scope**. This term describes the scope of protection for a Web service message. Protection scope refers to the extent the message will be protected, whether it is for its entire message lifetime or only while it is in transit between servers. This is also used as a category to describe transport layer security and message layer security in Table 3.1.

Transport Layer vs. Message Layer Security

Transport layer security represents an approach where the underlying operating system or application servers are used to handle security features. For data confidentiality, Secure Sockets Layer (SSL) is a common transport layer approach that is used to provide encryption. Figure 3.3 illustrates transport layer security.

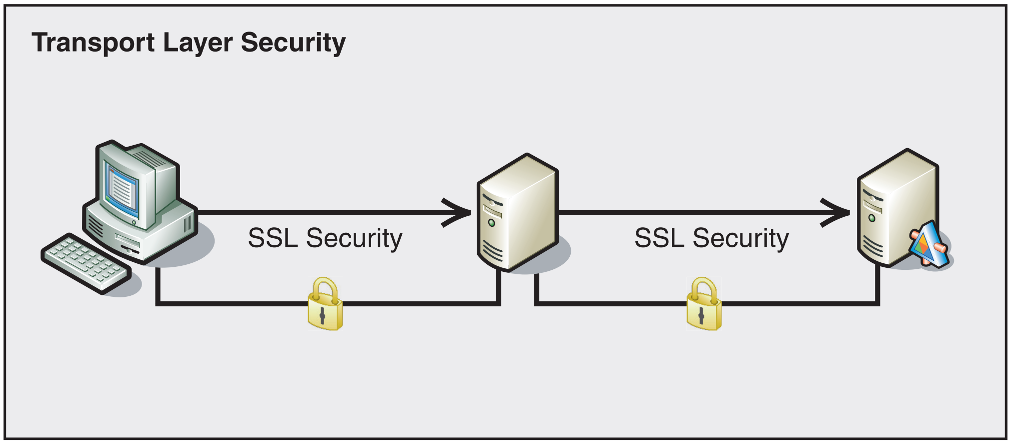


Figure 3.3
Transport layer security

If a message needs to go through multiple points to reach its destination, each intermediate point must forward the message over a new SSL connection. In this model, the original message from the client is not cryptographically protected on each intermediary because it traverses intermediate servers and additional computationally expensive cryptographic operations are performed for every new SSL connection that is established. Figure 3.4 illustrates message layer security.

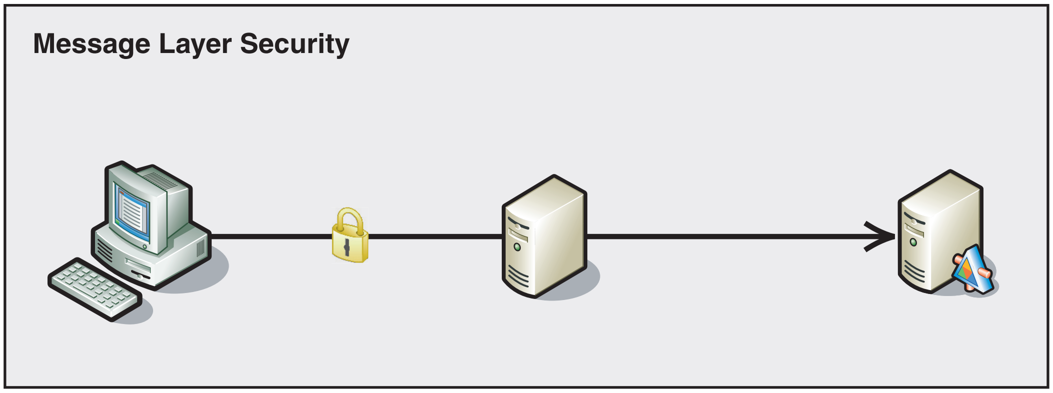


Figure 3.4
Message layer security

Message layer security represents an approach where all the information related to security is encapsulated in the message. Securing the message using message layer security instead of using transport layer security has several advantages that include:

- **Increased flexibility**. Parts of the message, instead of the entire message, can be signed or encrypted. This means that intermediaries can view the parts of the message that are intended for them. An example of this is a Web service that routes a SOAP message and is able to inspect unencrypted parts of the message to determine where to send the message, while other parts of the message remain encrypted. For an example of this, see the Perimeter Service Router pattern in Chapter 6, "Service Deployment Patterns."
- **Support for auditing**. Intermediaries can add their own headers to the message and sign them for the purpose of audit logging.
- **Support for multiple protocols**. You can send secured messages over many different protocols such as Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), and Transmission Control Protocol (TCP) without having to rely on the protocol for security.

Table 3.1 shows a decision matrix that lists security considerations related to protection scope and how each one is supported by transport or message layer security.

Table 3.1: Protection Scope Decision Matrix

Security consideration	Message layer	Transport layer
Your application interacts directly with the Web service.	Message layer protection is usually more CPU intensive than transport layer protection.	Transport layer HTTPS provides full message protection.
Web services are hosted on a system that does not support Windows Integrated Security.	Authentication can be performed by passing credentials in the message.	Basic over HTTPS could be implemented. However, it would require manipulation of message headers.
Your company has a firewall in place between applications and Web services.	Message layer security is not affected by standard firewalls.	It is not uncommon for port 443 to be opened to support HTTPS.
You have nonrepudiation requirements.	Supports persistence of messages that include digital signatures, which can be used to support nonrepudiation requirements.	You can use authentication with X.509 client certificates to support nonrepudiation.
A Web service request can pass through message queues or routing servers.	Message data will be protected as it passes through intermediate servers.	The message data is not protected as it passes through the server, which leaves it vulnerable to attack. With message queues in particular, it is possible that a decrypted message will be persisted until a dependent application retrieves the message.
Web services may require support for multiple protocols, including SMTP, FTP, HTTP, and TCP.	You can apply message layer protection to messages independent of the protocol that you used for transporting the message.	Different protocols have different built-in mechanisms to support security, making it difficult to standardize how services are secured.
The Web service you are designing will handle a high concurrent load.	You can use security tokens to establish a session. However, message protection is usually more CPU intensive.	You can use hardware appliances to improve performance with transport layer message protection protocols, such as SSL.

Table 3.1 lists some of the major security considerations you should examine when deciding between message and transport layer security.

For more information on implementing message layer security, see the following composite patterns:

- Implementing Direct Authentication with UsernameToken in WSE 3.0
- Implementing Message Layer Security with Kerberos in WSE 3.0
- Implementing Message Layer Security with X.509 Certificates in WSE 3.0

There is already a lot of good information available on using transport layer security to secure Web services, so this information is provided in the form of the following references, which point you to appropriate guidance for implementing transport layer security. For more information on implementing transport layer security, see the following sections in References for Transport Layer Security:

- Implementing Brokered Authentication Using Windows Integrated Security on IIS
- Implementing Transport Layer Data Confidentiality Using HTTPS
- Implementing Transport Layer Security Using HTTPS Basic over HTTPS
- Implementing Transport Layer Security Using X.509 Certificates and HTTPS
- Implementing Transport Layer Security with Kerberos and IPSec on Windows Server 2003

Implementing Direct Authentication with UsernameToken in WSE 3.0

Context

You are implementing direct authentication for an online application that consumes a Web service that uses Web Service Enhancements (WSE) 3.0. You are using message layer authentication. The credentials used to prove the identity of the client are validated by an authentication service.

Objectives

The objectives of this pattern are to:

- Implement direct authentication against Active Directory, Active Directory Application Mode (ADAM), or a custom SQL Server™ database using a security token that contains a user ID and password.
- Secure the communication channel by providing data confidentiality and data integrity. You can do this either at the message layer or at the transport layer.
- Demonstrate how to develop a custom **UsernameTokenManager** to support authentication against ADAM or a custom SQL database.
- Demonstrate how to use ASP.NET 2.0 membership providers for SQL Server and a directory service.

Content

This pattern consists of the following sections:

- **Implementation Strategy**: This section provides a high-level description of the strategy used to implement the Direct Authentication pattern. The section also discusses identity stores that you can use and different approaches to ensure secure communication between the participants.
- **Implementation Approach**: This section describes the steps necessary to implement this pattern:
  - General setup
  - Configure the client
  - Configure the service
- **Resulting Context**: This section outlines the benefits, liabilities, and security considerations related to this pattern.
- **Variants**: This section describes alternate choices to using Active Directory as an identity store, demonstrating how to implement both a database and a directory service as an identity store.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

The WSE 3.0 implementation of **UsernameToken** is used to implement direct authentication at the message layer. The client passes the credentials to the Web service as part of a secure message exchange. A password is sent in the message as plaintext, which is data in its unencrypted or decrypted form. The Web service decrypts the message, validates the credentials, verifies the message signature, and then sends an encrypted response back to the client.

Identity Store Options

There are three options for this pattern to implement different types of identity stores that the service can use to validate the credentials presented in a **UsernameToken**:

- Active Directory
- Database
- Directory service

Note: Direct authentication using Active Directory is described in the base pattern. The other two options are described at the end of this pattern as variants.

Active Directory

The ability to validate credentials presented in a **UsernameToken** to an Active Directory domain is provided with the **UsernameTokenManager** in WSE 3.0. Using Active Directory as an identity store has the following advantages:

- Unlike validating credentials using a database or a Lightweight Directory Access Protocol (LDAP)-enabled directory service, credential validation using Active Directory does not require a custom **UsernameTokenManager** class or an ASP.NET 2.0 membership provider.
- Of the three approaches for this pattern, Active Directory is the simplest option to implement in WSE 3.0.
- While Active Directory does require that users and their roles are maintained in an Active Directory infrastructure so that the service can use them to validate credentials, it does allow you to authenticate users without using Windows Integrated Security.

Database

You can use a database to store credentials that the service can then validate. Using a database as an identity store has the following advantages:

- It provides the capability to integrate with an existing database that is being used as an identity store. If you use a custom database schema, it may require you to implement a custom ASP.NET membership and possibly a role provider. For more information about how to create a custom identity provider, see the "Variant 2 — Using an LDAP Directory Service as the Identity Store" section later in this pattern.
- It supports transactional and concurrent updates to user credentials. For example, concurrent updates to security claims (such as role information for a single user) could occur if the maintenance of user credentials in the database is delegated to several different individuals. If concurrent updates are a concern, you should use either a directory service that supports transactional updates or a database to store user credentials and roles.

Using a database as an identity store does have the disadvantage that it is more difficult to maintain if the database is not shared across multiples services that authenticate the same users. This may cause data ownership and synchronization issues when changes are made to one identity store that must be propagated to the others.

For more information about using a database as an identity store, see the "Variant 1 — Using a Database as the Identity Store" section later in this pattern.

Directory Service

You also can use an LDAP-enabled directory service to store credentials for validation by the service. Using a directory service has the following advantages:

- It provides a viable alternative when you have an LDAP-enabled directory service in place of an Active Directory infrastructure.
- It can be used when you need to authenticate users using ADAM or Active Directory through LDAP ports due to firewall restrictions.

For more information about using a directory service as an identity store, see the "Variant 2 — Using an LDAP Directory Service as the Identity Store" section later in this pattern.

Providing Secure Communication

This implementation provides examples that show how to secure the communication channel between the client and the service, using both the **usernameForCertificateSecurity** and the **usernameOverTransportSecurity** WSE 3.0 turnkey assertions. The communication channel is secured by providing data confidentiality to prevent eavesdropping. Data origin authentication is also provided to prevent tampering or message spoofing. For more information, see Data Confidentiality and Data Origin Authentication in Chapter 2, "Message Protection Patterns."

The **usernameForCertificateSecurity** turnkey assertion secures the communication channel between the client and the service at the message layer using the service's X.509 certificate. But it is not compatible with client computers that have implemented WS-Security 1.0. This is because the **usernameForCertificateSecurity** turnkey assertion depends on the ability to reference **<EncryptedKey>** elements as security tokens, and enables the option for signature confirmation to correlate a response message with the request that prompted it. Both of these features are only available in WS-Security 1.1.

The **usernameOverTransportSecurity** turnkey assertion assumes that communication between the client and service will be secured at the transport layer. This approach is WS-Security 1.0 compatible, but it does not provide security features at the message layer. It also does not ensure that the channel is secured at the transport layer.

If you need to secure the communication channel between the client and service at the message layer with a solution that is compatible with WS-Security 1.0, you will need to create a custom policy assertion.

Note: At the time this pattern was published, most vendors supported WS-Security 1.0 implementations. WSE 3.0 supports features in WS-Security 1.1 and WS-Security 1.0. If you need to interoperate with platforms that do not support WS-Security 1.1 features, choose an option that best supports your interoperability requirements.

Participants

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to making access control decisions.
- **Identity store**. The entity that stores a client's credentials for a particular identity domain.

Process

The process section of Direct Authentication in Chapter 1, "Authentication Patterns," describes how identity and proof-of-possession are used for authentication. This pattern provides a more refined description of that process within the context of the implementation.

Figure 3.5 illustrates the direct authentication process.

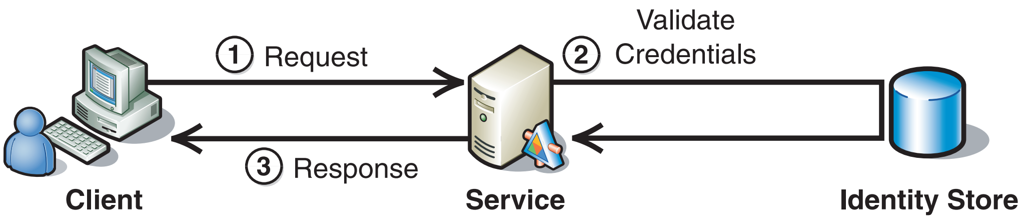


Figure 3.5
The direct authentication process

The steps for this implementation are divided into two parts, based on what happens with the client and what happens with the service:

- The client generates a Web service request.
- The service authenticates a client and returns a response.

The Client Generates a Web Service Request

This part of the process includes three steps:

1. Initialize the **UsernameToken**.
2. Establish message integrity.
3. Encrypt sensitive data in the message.

Step One: Initialize the UsernameToken

This pattern implements a **UsernameToken** with the **SendPlainText** password option to send the password over the network as plaintext. The plaintext value is the actual password because Active Directory requires plaintext passwords for credential validation. This option, which the default implementation of **UsernameTokenManager** uses, is similar to basic authentication over HTTP. You should always secure the communication between the client and server, either at the transport layer using Secure Sockets Layer (SSL) or at the message layer with WSE 3.0.

Step Two: Establish Message Integrity

Data origin authentication is established between the client and the service, either implicitly or explicitly, depending upon one of the two following methods that you can choose to secure messages between the client and the service:

- The **usernameOverTransportSecurity** turnkey assertion with HTTPS.
- The **usernameForCertificateSecurity** turnkey assertion.

HTTPS using the **usernameOverTransportSecurity** turnkey assertion provides data confidentiality and data integrity when you use server certificates. If you require data origin authentication from the client, you need to install and use a certificate for the client. For more information, see the reference, Implementing Transport Layer Security Using X.509 Certificates and HTTPS in Chapter 3, "Implementing Transport and Message Layer Security."

WSE 3.0 policy provides data confidentiality and data origin authentication when the **usernameForCertificateSecurity** assertion is used. The client includes a derived key token in the request message that is encrypted with a wrapped symmetric encryption key. The wrapped symmetric key is encrypted with the service's X.509 certificate public key. This key is referred to as an encrypted key. Accompanied by a valid **UsernameToken**, data origin authentication is provided when the client uses the derived key token to sign the message. For more information about derived key tokens, see Web Services Secure Conversation Language (WS-SecureConversation).

Step Three: Encrypt Sensitive Data in the Message

You should encrypt the message from the client to the service to ensure that only the service, as the intended recipient of the message, can process it. The method that you choose to secure the communication channel between the client and the service should also provide data confidentiality.

The Service Authenticates the Client and Returns a Response

This part of the process has five steps:

1. Decrypt the request message.
2. Verify message integrity.
3. Validate the password.
4. Establish the response integrity
5. Encrypt the response.

Step One: Decrypt the Request Message

The option you choose to secure communication between the client and the service determines how the request message is decrypted. The **usernameOverTransportSecurity** assertion relies on SSL to decrypt the message at the transport layer. WSE 3.0 policy using the **usernameForCertificateSecurity** assertion decrypts the derived key token encrypted with the wrapped symmetric key, and then uses the derived key token to decrypt the message signature, **UsernameToken**, and any other message parts that the client encrypted.

Step Two: Verify Message Integrity

The option you chose to secure communication between the client and the service determines how the message integrity is established and verified. The **usernameOverTransportSecurity** assertion relies on SSL to verify message integrity. If a client certificate is used for the client, the client also provides data origin authentication. WSE 3.0 using the **usernameForCertificateSecurity** assertion verifies the message integrity using the derived key token sent by the client that was decrypted in Step One.

Step Three: Validate the Password

After the service receives the message, the information in **UsernameToken** is verified by WSE 3.0 using the **UsernameTokenManager** class. WSE 3.0 uses the **AuthenticateToken** method of the **UsernameTokenManager** class to validate the information in the **UsernameToken**.

The **UsernameTokenManager** released with WSE 3.0 validates credentials against an Active Directory domain controller. If either a directory service or a database is used to store credentials for validation, then you will need to implement a custom **UsernameTokenManager** class. For more information, see the "Variants" section later in this pattern.

The **UsernameTokenManager** validates the username and password that was sent in the message with Active Directory through the **AuthenticateToken** method. The default **UsernameTokenManager** also establishes a **WindowsPrincipal** instance for the authenticated client and attaches it to the token's **Principal** property.

Step Four: Establish the Response Integrity

The method used to establish the response message's integrity depends upon whether communication is secured at the message layer using WSE 3.0 or at the transport layer using SSL. If communication is secured at the transport layer, message integrity is provided through SSL. If communication is secured at the message layer, the response message is signed with a key derived from the encrypted key that was sent in the request message.

Step Five: Encrypt the Response

The method used to encrypt the response message depends upon whether communication is secured at the message layer through WSE 3.0 or at the transport layer using SSL. If communication is secured at the transport layer, the response message is encrypted through SSL. If communication is secured at the message layer, the response signature and message parts are encrypted with a key derived from the encrypted key sent in the request message.

Implementation Approach

This section describes how to implement the pattern. This section is divided into three major tasks:

1. **General setup**. This task provides the required steps for both the client and the service.
2. **Configure the client**. This task provides the required steps to configure policy and code on the client.
3. **Configure the service**. This task provides the required steps to configure policy and code on the service.

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

General Setup

You must install WSE 3.0 on computers that you use to develop WSE-enabled applications. After WSE 3.0 is installed, you must enable the client and the service to support WSE 3.0. You can achieve this by performing the following steps.

► **To enable a Visual Studio 2005 project to support WSE 3.0**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable this project for Web Services Enhancements** check box, and then click **OK**.

If you are using the **usernameForCertificateSecurity** assertion to secure communication at the message layer between the client and service, you must configure the X.509 settings for WSE 3.0. For more information about setting up X.509 in WSE 3.0, see General Setup in the Implementing Message Layer Security with X.509 Certificates in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Note: WSE 3.0 offers four different protection levels that determine how messages are secured using SOAP message security. Generally, you should use the **Sign, Encrypt, and Encrypt Signature** setting for best message protection. This setting encrypts the message body and the XML signature, which reduces the likelihood of a successful cryptographic guessing attack against the signature. For this reason, all the composite implementation patterns use this value as default. If you want to use this setting in new Web services you should change the **messageProtectionOrder** attribute to the following value in your security policy:

```
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
```

Configure the Client

After enabling the client application to support WSE 3.0 during General Setup, you must enable policy support for it. If your application does not currently have a policy cache file, you can add one for this purpose, and enable policy support by performing the following steps.

► **To add policy support to a WSE 3.0-enabled Visual Studio 2005 project**

1. In Visual Studio 2005, right-click the application project and select WSE Settings 3.0.
2. On the **Policy** tab, select the **Enable Policy** checkbox. Selecting this setting adds a policy cache file with the default name *wse3policyCache.config*.
3. Under **Edit Application Policy**, click **Add**, and then type a policy friendly name for the new application policy, such as "usernameTokenSecurity."
4. Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides a choice to secure a service or a client. Select **secure a client application** to configure the client.
6. The wizard also provides a choice of authentication methods in the same step. Select **Username**, and then click **Next**.
7. On the **Optionally Provide Username and Password** page, the wizard provides you with options to define a user name and password. Ensure that the **Specify Username Token in code** checkbox is selected and click **Next**.

8. On the **Message Protection** page, you configure options for message protection. For transport layer security, select **None (rely on transport protection)** for the **Protection Order** to use the **usernameOverTransportSecurity** assertion. If you select any other protection option, the policy assertion will be **usernameForCertificateSecurity**.

   You should select the option for **Sign, Encrypt, Encrypt Signature**. By default, the **Enable WS-Security 1.1 Extensions** check box is enabled. This setting must be enabled if you are using message layer security. For more information about these settings, see the "Implementation Strategy" section earlier in this pattern.
9. Click **Next**.
10. If you selected **None (rely on transport protection)** to use transport security in Step 8, skip this step. If you selected any other option, the wizard will prompt you to select a server X.509 certificate for the service on the **Server Certificate** page. Change the Store Location to **LocalMachine** instead of using the default value of **CurrentUser**. Select the certificate for the service to use, and then click **Next**.
11. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete these tasks, your client security policy should look similar to the following code example. Examples for both the **usernameForCertificateSecurity** and **usernameOverTransportSecurity** assertions are included.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
 <extensions>
 </extensions>

 <!--Uncomment this policy to use the UsernameForCertificateSecurity scenario-->
 <policy name="usernameTokenSecurity">
  <usernameForCertificateSecurity establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="true" ttlInSeconds="60">
   <serviceToken>
    <!-- WSE2 QuickStart Server Certificate -->
    <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartServer" findType="FindBySubjectDistinguishedName" />
   </serviceToken>
   <protection>
    <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
    <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
    <fault signatureOptions="IncludeAddressing, IncludeTimestamp, IncludeSoapBody"
encryptBody="false" />
   </protection>
  </usernameForCertificateSecurity>
  <requireActionHeader />
 </policy>
```

(continued)

(continued)

```
 <!--Uncomment this policy to use the UsernameOverTransportSecurity scenario-->
 <!--<policy name="usernameTokenSecurity">
  <usernameOverTransportSecurity />
  <requireActionHeader />
 </policy>-->
</policies>
```

When you add a Web reference to the service from the client application, two proxies are generated for the Web service — one is a non-WSE 3.0 proxy and the other is WSE 3.0–enabled. In this guidance, Microsoft uses the WSE 3.0–enabled proxy class, which is defined as name + “Wse.” For example, if your Web service is named “MyService,” your WSE 3.0–enabled Web service proxy class name would be “MyServiceWse.”

The following code example provides an example of how to initialize an instance of a **UsernameToken** and to bind the appropriate policy defined in the preceding policy file to the Web service proxy. You can copy and insert this code into a new code module.

```
...
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
...
try
{
  Service.ServiceWse proxy = new Service.ServiceWse();
  string userName = null;
  if (txtDomain.Text.Trim().Length > 0)
  {
   userName = String.Format(@"{0}\{1}", txtDomain.Text, txtUsername.Text);
  }
  else
  {
   userName = txtUsername.Text;
  }

  UsernameToken token = new UsernameToken(userName, txtPassword.Text,
PasswordOption.SendPlainText);

  proxy.SetClientCredential(token);

  proxy.SetPolicy("usernameTokenSecurity");

  Service.Product product = proxy.GetProductInformation(txtProduct.Text);

  lblResults.Text = String.Format(CultureInfo.InvariantCulture,
            "Product: {0}, Quantity {1}, Unit price {2}",
            product.Name, product.Quantity, product.UnitPrice);
```

(continued)

continued)

```
}
catch (Exception ex)
{
  lblResults.Text = ex.ToString();
}
...
```

As appropriate, replace the **Product** class and code that processes the response returned from the service used in the preceding code example for the object type returned by your service.

Configure the Service

You must perform the following steps to configure the service to enable WSE 3.0 extensions.

► **To enable a Visual Studio 2005 project to support WSE 3.0 SOAP extensions**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable Microsoft Web Services Enhancement SOAP Protocol Factory** check box, and click **OK**.

After you enable the service application to support WSE 3.0 SOAP extensions, you must enable policy support. If your application does not currently have a policy cache file, you can add one and enable policy support by performing the following steps.

► **To add policy support to a WSE 3.0-enabled Visual Studio 2005 project**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **Policy** tab, select the **Enable Policy** check box. Selecting this check box adds the wse3policyCache.config file as the default name for the policy cache file.
3. Under **Edit Application Policy**, click **Add** and then type a policy friendly name for the new application policy, such as "usernameTokenSecurity."
4. Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides you with options to secure a service or a client. Select the **secure a service application** option button to configure the service.
6. The wizard also provides you with authentication method choices on the same page. Select **Username** and click **Next**.

7. On the **Users and Roles** page, you configure authorization based on the user name or roles associated with the user represented in the **UsernameToken**. by default, the **perform authorization** check box is cleared. If you want to perform authorization through the policy assertion, select the **perform authorization** check box, add users and roles as appropriate, and then click **Next**.
8. On the **Message Protection** page, you configure options for message protection. For transport layer security, select **None (rely on transport protection)** for the **Protection Order** to use the **usernameOverTransportSecurity** assertion.

   If you select any other protection option, the policy assertion will use **usernameForCertificateSecurity**. If you select any option under **Protection Order** other than **None (rely on transport protection)**, select the option for **Sign, Encrypt, Encrypt Signature**.

   By default, the **Enable WS-Security 1.1 Extensions** check box is selected. You must enable this option if you are using certificate security. For more information about these settings, see the "Implementation Strategy" section earlier in this pattern.
9. Click **Next**.
10. If you opted to use transport security by selecting the **None (rely on transport protection)** setting in step 8, skip this step. If you selected any other option, the wizard will prompt you to select a server X.509 certificate for the service on the **Server Certificate** page. Select the certificate that you want to use for the service, click **Next**.
11. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete these tasks, your client security policy should look similar to the following code example. Examples for both the **usernameForCertificateSecurity** and **usernameOverTransportSecurity** assertions are included.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
 <!--Uncomment this policy to use the UsernameForCertificateSecurity scenario-->
 <policy name="usernameTokenSecurity">
  <authorization>
   <allow role="Users" />
   <deny role="*" />
  </authorization>
  <usernameForCertificateSecurity establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="true" ttlInSeconds="60">
   <serviceToken>
    <!-- WSE2 QuickStart Server Certificate -->
    <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartServer" findType="FindBySubjectDistinguishedName" />
   </serviceToken>
   <protection>
    <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
```

(continued)

(continued)

```
    <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
    <fault signatureOptions="IncludeAddressing, IncludeTimestamp, IncludeSoapBody"
encryptBody="false" />
   </protection>
  </usernameForCertificateSecurity>
  <requireActionHeader />
 </policy>

 <!--Uncomment this policy to use the UsernameOverTransportSecurity scenario-->
 <!--<policy name="usernameTokenSecurity">
  <authorization>
   <allow role="Administrators" />
   <deny role="*" />
  </authorization>
  <usernameOverTransportSecurity />
  <requireActionHeader />
 </policy>-->
</policies>
```

The service's policy configuration is identical to the client's, except that the policy assertions for the service can contain an **<authorization>** assertion. This assertion allows users who belong to the Users group to call the service, and denies access to all other users. The roles that this policy assertion evaluates are obtained when the user is authenticated. The default **UsernameTokenManager** populates a security principal containing the user's roles in the Active Directory domain.

Note: WSE 3.0 uses the default **UsernameTokenManager** class to validate credentials presented in a **UsernameToken** by calling the Win32 **LogonUser** function. In Windows XP and Windows 2000, the service account, under which the Web application validating the credentials runs, can only call the **LogonUser** function if it has **Log on locally** permissions to the server hosting the service.

The following code example demonstrates how to apply the policy provided earlier when the service processes a request. You can copy and insert this code into a new code module.

```
using System;
using System.Web.Services;

using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security.Tokens;

using Microsoft.Practices.WSSP.WSE3.QuickStart.Common;
```

(continued)

(continued)

```
namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.UsernameTokenWithWindows.Service
{
  /// <summary>
  /// This class represents a web service used to query products catalog, secured
with a UsernameToken
  /// </summary>
  [WebService(Namespace =
"http://schemas.microsoft.com/WSSP/WSE3/QuickStart/DirectAuthentication/2005-
10/UsernameTokenWithWindows.wsdl")]
  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
  [Policy("usernameTokenSecurity")]
  public class Service : System.Web.Services.WebService
  {
    const string AdmistratorsRole = "Administrators";

    public Service()
    {
    }

    /// <summary>
    /// Returns some information about the specified product
    /// </summary>
    /// <param name="productName"></param>
    /// <returns></returns>
    [WebMethod]
    public Product GetProductInformation(string productName)
    {
      CheckPrincipalRoles();

      Product product = new Product();
      product.Name = productName;
      product.Quantity = 10;
      product.UnitPrice = 2.5M;
      return product;
    }

    /// <summary>
    /// Verifies if the user has permissions to execute this service
    /// </summary>
    private void CheckPrincipalRoles()
    {
      SecurityToken token = RequestSoapContext.Current.IdentityToken;
      bool isInRole = token.Principal.IsInRole(AdmistratorsRole);

      if (!isInRole)
      {
        throw new
UnauthorizedAccessException(string.Format(Resources.Messages.AuthorizationExceptio
n, AdmistratorsRole));
      }
    }
  }
}
```

In the preceding code example, the Web service applies the appropriate policy through the **Policy** attribute in the class declaration. Ensure that the value specified in the **Policy** attribute matches the name of your policy assertion that you want to use.

The **UnauthorizedAccessException** class uses a string from a resource file to provide a message for the exception. Alternatively, a simple string could be provided instead of accessing a resource file.

If you secure communication at the transport layer using the **usernameForCertificateSecurity** assertion, you must also install an X.509 certificate into the local machine certificate store where the service is hosted. Also, you must ensure that the service account under which the service is configured to run has read permissions to the certificate private key. You can do this by using the Certificates tool released with WSE 3.0. If you are running the service under the default service account for ASP.NET, you need to grant read permissions to that account. On Windows 2000 and Windows XP, the default account is ASPNET. On Windows Server 2003, the default account is the NETWORK SERVICE account.

When securing direct authentication using X.509 certificates either at the message layer or the transport layer, ensure that anonymous access is enabled for the virtual directory where the service is hosted in Internet Information Services (IIS) 6.0. Otherwise, the service may expect the client to authenticate at the transport layer and reject the client's attempts to authenticate at the message layer with a **UsernameToken**.

► **To enable a Anonymous Access on a virtual directory in IIS 6.0**

1. In IIS 6.0, right-click the virtual directory where the service is hosted, and then select **Properties**.
2. Click the **Directory Security** tab.
3. Under **Authentication and access control**, click **Edit**.
4. Ensure that the **Enable anonymous access** checkbox is selected, click **OK**, and then click **OK** again.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Implementing Direct Authentication with UsernameToken in WSE 3.0 pattern include the following:

- The pattern provides interoperable password-based authentication at the message layer.
- The pattern allows for flexibility to secure communication at either the message layer or the transport layer.
- The pattern enables flexible configuration for using different authentication services/identity stores to validate credentials presented in a **UsernameToken**.

Liabilities

The liabilities associated with the Implementing Direct Authentication with UsernameToken in WSE 3.0 pattern include the following:

- When using **UsernameTokens**, you can configure WSE 3.0 to prevent replay attacks by using a nonce and timestamp with a replay cache on the server through configuring the **<replayDetection>** element. For more information about this topic, see <replayDetection> Element. However, the replay cache is not shared across a server farm. One solution you can use to mitigate this issue is to create a replay cache that is shared across the server farm. If you are using the **usernameOverTransportSecurity** assertion, the method used to secure communication at the message layer (such as SSL) must provide message replay detection because the message is not signed. For more information about message replay detection, see Message Replay Detection and Implementing Message Replay Detection in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."
- The **usernameForCertificateSecurity** assertion uses features that are introduced in WS-Security 1.1, which makes it incompatible with Web services implementing the WS-Security 1.0 specification.
- Implementing message layer security is likely to reduce the throughput and increase the latency of Web services, due to the overhead of the cryptographic operations that support canonicalization, XML signatures, and encryption. As part of your development process, you should identify performance objectives for your application and test the application against those objectives. For more information, see *Improving .NET Performance and Scalability*.

Security Considerations

Security considerations associated with the Implementing Direct Authentication with UsernameToken in WSE 3.0 pattern include the following:

- The password in a **UsernameToken** should always be encrypted, using either message layer security or transport layer security, such as SSL. This mitigates the threat of an eavesdropper obtaining credentials from the **UsernameToken**.
- If SSL is implemented between several intermediaries providing point-to-point security, the environment is vulnerable to man-in-the-middle and XML attacks.

Passwords are considered one of the weakest forms of identity used for authentication, but they are also the most common. As a result, it is important to understand threats and vulnerabilities associated with passwords. Passwords are often based on words and phrases that users can remember. This makes it easier to discover passwords through using brute force attacks that try thousands of common passwords and word combinations. You can mitigate this vulnerability by using complex passwords or password phrases, although if user passwords become too difficult to remember, users are likely to write them down.

Variants

The following variants describe alternate choices to Active Directory as an identity store, as discussed in the "Identity Store Options" section under the Implementation Strategy section earlier in this pattern. Both the database and directory service identity stores require a custom **UsernameTokenManager** class and an ASP.NET 2.0 membership provider that is configured for them.

Variant 1 — Using a Database as the Identity Store

Instead of validating credentials with an Active Directory domain controller as described in the base pattern, this variant describes how to configure the implementation to use a database as the identity store.

As previously stated in this pattern, whenever you use something other than Active Directory to manage user credentials, WSE 3.0 requires you to use a custom **UsernameTokenManager** class and an ASP.NET 2.0 membership provider that is configured for the service. For instructions and examples about how to create and configure a custom **UsernameTokenManager** class, see "Create a Custom UsernameTokenManager" at the end of this section.

To use a database as an ASP.NET 2.0 membership provider, you must configure the service to use a **SqlMembershipProvider**. For more details about how to configure a **SqlMembershipProvider**, see "Using the SQLMemberShipProvider" in How To: Use Membership in ASP.NET 2.0. After following these steps to configure the **SqlMembershipProvider** for your service, the configuration for your membership provider should look similar to the following service's Web.config file.

```
...
<connectionStrings>
 <add name="MySqlConnection" connectionString="Data Source=MySqlServer;Initial
Catalog=aspnetdb;Integrated Security=SSPI;" />
</connectionStrings>
<system.web>
...
 <membership defaultProvider="SqlProvider" userIsOnlineTimeWindow="15">
 <providers>
  <clear />
  <add
  name="SqlProvider"
  type="System.Web.Security.SqlMembershipProvider"
  connectionStringName="MySqlConnection"
  applicationName="MyApplication"
  enablePasswordRetrieval="false"
  enablePasswordReset="true"
  requiresQuestionAndAnswer="true"
  requiresUniqueEmail="true"
  passwordFormat="Hashed" />
 </providers>
 </membership>
...
```

Variant 2 — Using an LDAP Directory Service as the Identity Store

Instead of validating credentials with an Active Directory domain controller as described in the base pattern, this variant describes how to configure the implementation to use a an LDAP-enabled directory service as an identity store.

As previously stated in this pattern, whenever you use something other than Active Directory to manage user credentials, WSE 3.0 requires you to use a custom **UsernameTokenManager** class and an ASP.NET 2.0 membership provider that is configured for the service. For instructions and examples about how to create and configure a custom **UsernameTokenManager**, see the end of this section.

To use Active Directory through LDAP or ADAM joined to an Active Directory instance, you must configure the service to use an **ActiveDirectoryMembershipProvider**. For more details about how to configure an ASP.NET 2.0 membership provider, see How To: Use Membership in ASP.NET 2.0.

After following these steps to configure the **ActiveDirectoryMembershipProvider** for your service, the configuration for your membership provider should look similar to the following service's Web.config file. The connection string in this code example has been substituted for the one that is required to connect your directory service. An ellipsis (...) represents sections of the configuration file that have been omitted for brevity.

```
<connectionStrings>
 <add name="ADConnectionString"
 connectionString=
 "LDAP://domain.testing.com/CN=Users,DC=domain,DC=testing,DC=com" />
</connectionStrings>
...
<system.web>
 ...
 <membership defaultProvider="MembershipADProvider">
 <providers>
 <add
  name="MembershipADProvider"
  type="System.Web.Security.ActiveDirectoryMembershipProvider, System.Web,
   Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    connectionStringName="ADConnectionString"
    connectionUsername="<domainName>\directoryservice"
    connectionPassword="password"/>
 </providers>
 </membership>
 ...
</system.web>
...
```

Different directory services may require different formatting of a user name when credentials are validated. For example, ADAM requires a format of *username@domain*. The client can do this when it creates a **UsernameToken** instance. In which case, the service should check the formatting in the **CustomUsernameTokenManager** before the credentials are validated against the directory service. The formatting can also be done directly in the **CustomUsernameTokenManager** before the credentials are validated against the directory service, with the expectation that the client will send the user name without a specified domain, and that the **CustomUsernameTokenManager** will add the domain name with proper formatting.

If you use an LDAP-enabled directory service other than Active Directory or ADAM to validate credentials, you may need to create a custom membership provider. For more details on how to build custom ASP.NET 2.0 providers, see Building Custom Providers for ASP.NET 2.0 Membership. Also, depending how you store and retrieve account roles in your directory service, you may need to implement a custom **RoleProvider**. For example, if you use an LDAP schema for user roles that is not supported through **ActiveDirectoryMembershipProvider**, you will need to implement a custom **RoleProvider** to retrieve roles for your users.

In a custom **RoleProvider** class, you need to retrieve the user roles from the directory service by overriding the **GetRolesForUser()** method. The code to retrieve user roles from the directory service would look like the following example.

```
public override string[] GetRolesForUser(string username)
    {
      using (DirectoryEntry rootEntry = new DirectoryEntry(this.connectionString))
      {
        rootEntry.Username = this.username;
        rootEntry.Password = this.password;

        rootEntry.AuthenticationType = AuthenticationTypes.None;
        rootEntry.RefreshCache();

        //Search the user in the directory service
        using (DirectorySearcher searcher = new DirectorySearcher(rootEntry))
        {
          searcher.PropertiesToLoad.Add("memberOf");
          searcher.PropertiesToLoad.Add(this.usernameAttribute);

          searcher.Filter = String.Format("(&(objectClass=user)({0}={1}))",
this.usernameAttribute, username);
          SearchResult result = searcher.FindOne();
          DirectoryEntry userEntry = result.GetDirectoryEntry();

          string[] roles = null;

          PropertyValueCollection property = userEntry.Properties["memberOf"];
          if (property.Value is Array)
          {
            Array values = (Array)property.Value;
            roles = new string[values.Length];
            values.CopyTo(roles, 0);
          }
          else if (property.Value is string)
          {
            roles = new string[1];
            roles[0] = (string)property.Value;
          }
          return roles;
        }
      }
    }
```

Create a Custom UsernameTokenManager

When validating credentials against a database or an LDAP-enabled directory service, you need to create and implement a custom **UsernameTokenManager** class. This is not necessary if you are validating credentials against an Active Directory domain.

To implement a custom **UsernameTokenManager** for either a database or a directory service, you must derive a custom class from the **UsernameTokenManager** and configure the service to use the custom class in its Web.config file.

The easiest way to add an entry for a custom **UsernameTokenManager** in the service's Web.config file is by using the WSE 3.0 Settings tool. To add a custom **UsernameTokenManager** entry, right-click the service project, select **WSE Settings 3.0**, and then on the **Security** tab, type the security token manager's information.

The following configuration example provides an example of what a custom **UsernameTokenManager** in the service's Web.config file might look like after you have added it through the WSE 3.0 Settings tool. An ellipsis (...) indicates configuration sections that have been omitted for brevity.

```
<configuration>
...
  <microsoft.web.services3>
   ...
   <securityTokenManager>
     <add localName="UsernameToken"
type="Microsoft.Practices.WSSP.WSE3.QuickStart.UsernameTokenWithDatabase.Service.C
ustomUsernameTokenManager" namespace="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"/>        ...
    </securityTokenManager>
      <Microsoft.web.services3>
   ...
</configuration>
```

In the previous example, the **type** attribute represents the fully qualified name of the custom **UsernameTokenManager** class. Set this attribute based on the namespace and class name that you chose for your custom **UsernameTokenManager** class.

The following code example provides an example of a custom **UsernameTokenManager** class.

```
using System;
using System.Xml;
using System.Security.Permissions;
using System.Web.Security;
using System.Security.Principal;

using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
```

(continued)

(continued)

```
namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.UsernameTokenWithDatabase.Service
{
    /// <summary>
    /// By implementing UsernameTokenManager we can verify the signature
    /// on messages received.
    /// </summary>
    [SecurityPermissionAttribute(SecurityAction.Demand,
Flags=SecurityPermissionFlag.UnmanagedCode)]
    public class CustomUsernameTokenManager : UsernameTokenManager
    {
        /// <summary>
        /// Constructs an instance of this security token manager.
        /// </summary>
        public CustomUsernameTokenManager()
        {
        }

        /// <summary>
        /// Constructs an instance of this security token manager.
        /// </summary>
        /// <param name="nodes">An XmlNodeList containing XML elements from a
configuration file.</param>
        public CustomUsernameTokenManager(XmlNodeList nodes)
            : base(nodes)
        {
        }

        /// <summary>
        /// Returns the password or password equivalent for the username provided.
    /// Adds a principal to the token with user's roles.
        /// </summary>
        /// <param name="token">The username token</param>
        /// <returns>The password (or password equivalent) for the
username</returns>
        protected override string AuthenticateToken( UsernameToken token )
        {
      bool validCredentials = Membership.ValidateUser(token.Username,
token.Password);
      if (!validCredentials)
      {
        throw new ApplicationException(Resources.Messages.AuthenticationError);
      }

      GenericIdentity identity = new GenericIdentity(token.Username);
      GenericPrincipal principal = new GenericPrincipal(identity,
Roles.GetRolesForUser(token.Username));
      token.Principal = principal;

      return token.Password;
        }

    }
```

Implementing Message Layer Security with Kerberos in WSE 3.0

Context

You are implementing brokered authentication in an application deployed on computers running Windows with security implemented at the message layer. A Web service using Web Services Enhancements (WSE) 3.0 is processing messages from clients. The clients and services must use a standards-based security token that uses the organization's existing Active Directory infrastructure. The solution must be able to provide a complete set of security features, including data origin authentication and data confidentiality.

Objectives

The objectives of this pattern are to:

- Use an existing infrastructure that employs the Kerberos version 5 protocol at the message layer with a **KerberosToken** binary security token.
- Secure the communication channel to provide data confidentiality and data integrity by encrypting and signing messages with the **KerberosToken**.
- Impersonate authenticated clients that the **KerberosToken** represents to access resource on their behalf. A client can be a user, application, or server that needs to be authenticated before it can access a service.

Content

This pattern consists of the following sections:

- **Implementation strategy**. This section provides a high-level description of the strategy used to implement the solution that includes a description of the participants and the process.
- **Implementation approach**. This section describes the steps necessary to implement this pattern:
  - General setup
  - Client setup
  - Service setup
- **Resulting context**. This section outlines the benefits, liabilities, and security considerations related to this pattern.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

Use an existing Kerberos infrastructure, such as the one in Active Directory to provide authentication and access control on client workstations and servers that host Web applications. Use the **kerberosSecurity** policy assertion in WSE 3.0 to provide authentication, data confidentiality, and integrity at the message layer. For more information about WSE 3.0 policy, see Securing a Web Service on MSDN. This implementation also demonstrates how to use a **KerberosToken** to establish a Windows security context. The service then calls a second service that is configured for Windows Integrated Security.

Participants

Message layer security with the Kerberos protocol in WSE 3.0 involves the following participants:

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.
- **Key Distribution Center (KDC)**. The KDC is the authentication broker that authenticates clients and issues service tickets. On the Windows platform, the KDC is implemented in Active Directory.

Process

The “Process” section of Brokered Authentication: Kerberos in Chapter 1, “Authentication Patterns” describes how you can use a **KerberosToken** security token for message layer authentication with a Web service. The session keys created during Kerberos authentication also can sign and encrypt messages. These capabilities allow you to implement data origin authentication and data confidentiality as part of the authentication process. As a result, this pattern includes additional steps to represent a complete message layer security solution that implements authentication, data origin authentication, and data confidentiality.

Note: Windows 2000 does not support **KerberosToken** for signing and encryption. For more information about this and other information related to the Kerberos protocol, see Kerberos Technical Supplement for Windows in Chapter 7, “Technical Supplements.”

This pattern provides a more detailed description of the implementation process that the design pattern describes. The steps are divided into the following two parts, based on what happens on the client and then on the service:

- The client initializes a Kerberos security token and sends it in a message to a service.
- The service authenticates the client using information found in the security token.

Client: Initialize the Security Token and Send the Message

The client performs the following five steps to complete this task:

1. Request a service ticket.
2. Retrieve the service ticket.
3. Sign the message.
4. Encrypt the message.
5. Send the message to the service.

The steps are summarized in Figure 3.6.

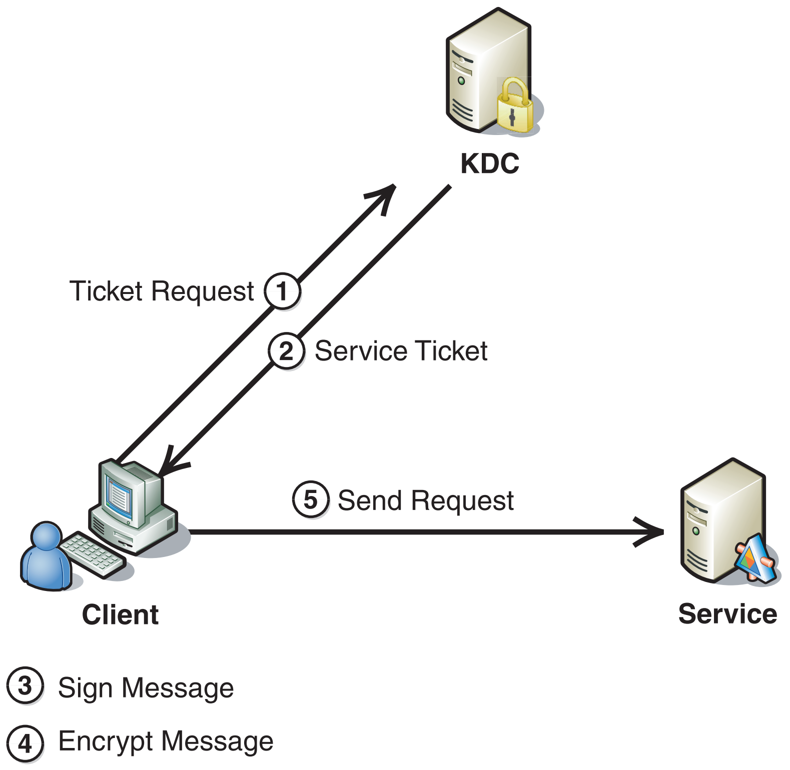


Figure 3.6
Initializing and sending a message using the Kerberos protocol

The following sections describe these steps.

Step One: Request a Service Ticket

The client interacts with the KDC to retrieve a service ticket, which it then uses to access the Web service. This action is actually performed as a request to the Security Support Provider Interface (SSPI) implementation in the Local Security Authority (LSA) to initialize a security context. The LSA accesses the client's ticket-granting ticket (TGT) and uses that to request a service ticket from the KDC. For more information about the Kerberos protocol implementation in Windows-based software, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

Step Two: Retrieve the Service Ticket

The ticket-granting service (TGS) creates a service ticket and returns it to the Local Security Authority (LSA). The LSA uses the service ticket to complete the initialization of a security context. WSE 3.0 uses the new security context to initialize a **KerberosToken** that is used to access the service. When a **KerberosToken** is initialized in WSE 3.0, two keys are derived from the session key in the service ticket for both the client and the service to use. One key is used to sign messages, and the other is used to encrypt them as described later in the process.

Step Three: Sign the Message

The message is signed using the security token retrieved in the previous step. You can choose to sign one or more portions of the message, such as the address header or the message body. An XML signature is created using a symmetric signature algorithm that computes a hash from the data to be signed using a signing key derived from the session key in the security token. When an XML signature is validated, the data used to create the signature is also validated to provide data origin authentication. At a minimum, you should include the addressing headers, timestamp, and message body in the message signature.

Step Four: Encrypt the Message

You can encrypt the message body using a security token that is derived from the session key in the security token. In addition, you should encrypt the message signature to reduce the risk of an offline cryptographic attack on the signature.

Step Five: Send the Message to the Service

After the client computer signs and encrypts the message, it sends the message to the service. When the message is sent, WSE 3.0 automatically adds the Kerberos security token to the message as a **BinarySecurityToken**.

Service: Authenticate the Client

There are five steps for the service to perform to complete this task:

1. Validate the token.
2. Decrypt the message.
3. Verify the XML signature.
4. Authorize and/or impersonate the client (optional).
5. Initialize and send a response to the client (optional).

The steps are summarized in Figure 3.7.

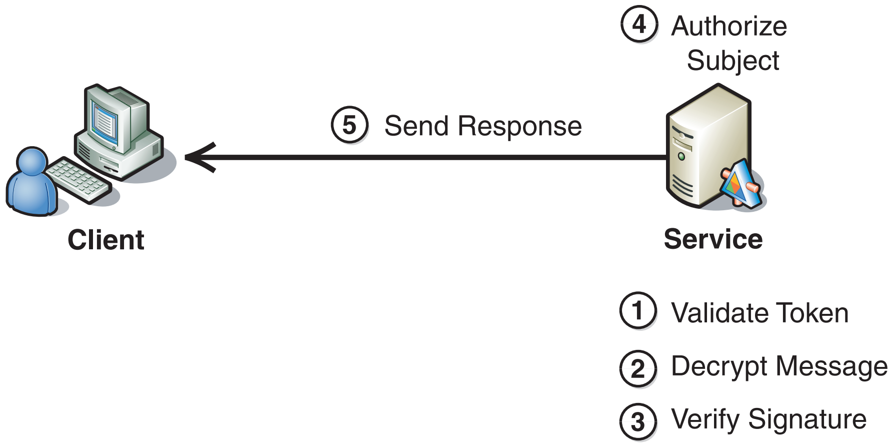


Figure 3.7
Authenticating a client using the Kerberos protocol

The following describes each of these steps in this section.

Step One: Validate the Token

When a service receives a message with a Kerberos security token, it needs to acquire credentials for the service account from the service host. To access the credentials, a service must be running under a process identity that has access to the service credentials. For more information about configuring the service identity with different operating systems, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

The service credentials contain the service's master key, which decrypts the service ticket in the message that the client sent. The service ticket contains a session key, which decrypts the authenticator and validates the message. For more information about Kerberos authenticators, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

After the service validates the message, it accepts the security token and uses the client's information in the service ticket to initialize a security context.

tep Two: Decrypt the Message

When WSE 3.0 receives a message that has been encrypted, WSE 3.0 policy does the following to decrypt the message:

1. Retrieve the symmetric session key from the service ticket.
2. Generate the derived encryption key from the session key.
3. Use the derived key to decrypt the message data with a symmetric algorithm.

Note: The policy on the server does not stop someone from sending an unencrypted message. However, it does reject a message at the server if it is not encrypted. The client can also implement a policy assertion that requires outbound messages to be encrypted.

Step Three: Verify the XML Signature

After the service receives the message. WSE 3.0 policy validates the message signature using the derived signing key that was sent with the message. This step validates the origin of that data to provide data origin authentication. However, note that XML signatures created using a symmetric algorithm do not support nonrepudiation. For more information about data origin authentication, see Data Origin Authentication in Chapter 2, "Message Protection Patterns."

Step Four: Authorize and/or Impersonate the Client (Optional)

By default, when WSE 3.0 receives a message that contains a Kerberos security token, it accesses the client's authorization claims that are contained in the service ticket. The Kerberos protocol allows these claims to perform authorization tasks, and the service ticket also can impersonate the client.

Step Five: Initialize and Send a Response to the Client Computer (Optional)

If the service returns a secure response to the client, the response must use the same security token that the request used. To accomplish this, the request message must be signed and encrypted using keys derived from the same token that was received in the request message from the client.

Implementation Approach

This section describes how to implement this pattern. The section is divided into three major tasks:

- General setup
- Client setup
- Service setup

Note: Applications using **KerberosTokens** will not function properly if they are hosted in Cassini. You must use Internet Information Services (IIS) 6.0 to host them. Cassini is a local access only Web server distributed with Visual Studio 2005 to allow Web development without IIS. One way to tell if a Web application is hosted in Cassini or in IIS is to look at the project in the Visual Studio 2005 solution explorer. If the Web application project appears as a file path (for example, C:\directory), Cassini is hosting the application. If the Web application project appears as an URL, IIS is hosting it.

General Setup

You must install the WSE 3.0 SDK on the computers that you use to develop WSE-enabled applications. After you have installed WSE 3.0, you must enable the client and the service to support WSE 3.0. You can achieve this by performing the following steps.

► **To enable a Visual Studio 2005 project to support WSE 3.0**

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable this project for Web Service Enhancements** check box, and then click **OK**.

You may be required to perform additional configuration steps to allow the ASP.NET process to access service credentials. You can use either of the following two approaches for this purpose. The approach to use depends on the Windows operating system that you used to install WSE 3.0:

- **Use the existing ASP.NET worker process**. This is the preferred option to use on computers running Windows Server 2003. No configuration is required in this case. The ASP.NET worker process uses a different account that has all of the necessary rights required to access service credentials.
- **Create a new domain account and map that account to the service host using setspn.exe**. This is the preferred option to use on computers running Windows XP and Windows 2000. To use this option, modify the Machine.config file and set the **userName** and **password** attributes to the new domain account in the **processModel** element, and then reset IIS 6.0. For more information about this option, see "Kerberos Operations for Web Services" in the Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

Caution: It is theoretically possible to configure the **processModel** element to use the SYSTEM account in a production environment. While this would give the ASP.NET process access to service credentials, using this account represents a serious security risk that could be catastrophic to your environment.

Note: WSE 3.0 offers four different protection levels that determine how messages are secured using SOAP message security. Generally, you should use the **Sign, Encrypt, and Encrypt Signature** setting for best message protection. This setting encrypts the message body and the XML signature, which reduces the likelihood of a successful cryptographic guessing attack against the signature. For this reason, all the composite implementation patterns use this value as default. If you want to use this setting in new Web services you should change the **messageProtectionOrder** attribute to the following value in your security policy:

```
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
```

Client Setup

This task requires the following steps to configure the client to implement this pattern:

1. **Configure the policy**. Includes the WSE 3.0 policy configuration settings necessary to implement this pattern on the client.
2. **Add the client code**. Includes the coding necessary to successfully implement this pattern on the client.

Configure the Policy

After enabling the client application to support WSE 3.0, you must enable policy support. If your application does not currently have a policy cache file, you can add one, and then perform the following procedure to enable policy support.

► **To add policy support to a WSE 3.0 enabled Visual Studio 2005 project**

1. In Visual Studio, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **Policy** tab, select the **Enable Policy** checkbox. Selecting this checkbox adds a policy cache file with the default name wse3policyCache.config.
3. Under **Edit Application Policy**, click **Add**, and then type a policy friendly name for the new application policy, such as “KerberosClient.”
4. Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides a choice to secure a service or a client. Select the option for **secure a client application** to configure the client.
6. The wizard also provides a choice of authentication methods in the same step. Select **Windows**, and then click **Next**.
7. On the **Kerberos Token** page, the wizard provides you with the option to provide a service principal name (SPN) and to specify the impersonation level for the Kerberos Token. The example for this pattern specifies the SPN as “http/server1.” Replace “server1” with the name of the target Web server for the service that you will use. Then select **Impersonation** for the impersonation level and click **Next**.

8. On the **Message Protection** page, the wizard provides you with configuration options for message protection. You should select the option for **Sign, Encrypt, Encrypt Signature**. By default, the **Enable WS-Security 1.1 Extensions** setting is selected. Ensure that the extensions are enabled if you want WSE 3.0 to use signature confirmation to correlate a response message from the service with the request message that prompted it.
9. If your primary concern is interoperability, clear the **Enable WS-Security 1.1 Extensions** check box.

   – or –

   If you want to use signature confirmation, leave the **Enable WS-Security 1.1 Extensions** check box selected, click **Advanced Settings**, ensure the **Enable signature confirmation** check box is selected, click **OK**, and then click **Next**.
10. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete the procedure, your client security policy cache should appear similar to the following code example.

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="KerberosClient">
    <kerberosSecurity establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="true" ttlInSeconds="300">
      <token>
        <!-- By default this sample does not work until you have changed the
TargetMachineName value -->
        <!-- Change the TargetMachineName value to the machine name with the Web
Service e.g. targetPrincipal="host/server1" -->
        <kerberos targetPrincipal="http/server1"
impersonationLevel="Impersonation" />
      </token>
      <protection>
        <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <fault signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="false" />
      </protection>
    </kerberosSecurity>
    <requireActionHeader />
  </policy>
</policies>
```

The **kerberosSecurity** policy assertion provides the ability to sign and encrypt messages using policy with a **KerberosToken** binary security token in WSE 3.0.

In the previous example, the policy is declared as **KerberosClient**. It contains an assertion called **<kerberosSecurity>**. The **requireSignatureConfirmation** attribute controls whether the policy uses signature confirmation to provide a correlation between a response and the request that prompted it.

The **messageProtectionOrder** attribute defines the order in which the policy signs and encrypts the message. As recommended in the previous "Process" section for this pattern, when the client computer signs a message, ensure that the message signature also is encrypted. Setting the value of the **messageProtectionOrder** attribute to **SignBeforeEncryptAndEncryptSignature** will provide the recommended behavior. For more information about Kerberos assertion policy settings, see <kerberosSecurity> Element on MSDN.

The **<token>** section of the assertion provides details about the Kerberos token. You must set the **targetPrincipal** attribute to "http/," and include the name of the server where the Web service is hosted. If you have created a custom SPN, ensure that its name appears here. The **impersonationLevel** attribute allows you to specify whether you want the token to identify the client or impersonate the client. In this pattern this attribute is set to **Impersonation** so that a resource can be accessed on the client's behalf.

Note: You can use the prefix "host/" instead of "http/" for the SPN. However, doing this eliminates the option to use Windows Integrated Security on the target to access additional resources on behalf of the client computer. For more information about SPNs, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

The **<protection>** section of the assertion allows you to specify protection options on the request, response, and fault messages using the **<request>**, **<response>**, and **<fault>** elements, respectively. The options available for each of these elements are the same. The **signatureOptions** attribute allows you to specify which parts of the message are signed in a comma separated list. To achieve the behavior recommended previously in the Process section for when the client signs the message, specify the **IncludeAddressing**, **IncludeTimestamp**, and **IncludeSoapBody** attributes in the value for the **signatureOptions** attribute. Setting the value of the **encryptBody** attribute to **true** encrypts request messages and decrypts response messages.

Add the Client Code

The following example code displays how to implement the client as a Web service client. It provides an example of binding the previously defined policy to the proxy, and then calling a Web service. You can copy the code to insert it into a new code module.

```
...
Service.ServiceWse service = new Service.ServiceWse();
service.SetPolicy("KerberosClient");

Product[] products = service.GetProductInformation(txtProductName.Text);

txtResults.Text = string.Empty;
foreach (Product product in products)
{
    txtResults.Text += String.Format( CultureInfo.InvariantCulture,
"Product Name: {0}, Unit Prize: {1}, Quantity: {2}",
        product.Name, product.UnitPrice, product.Quantity);
}
...
```

If the client is an ASP.NET application, you must configure your client application to use Windows authentication and set the **impersonate** attribute of the **<identity>** attribute to **true** in the client's Web.config file or programmatically impersonate the user in code. The following code example provides an example of how to configure security in the client's Web.config file.

```
<configuration>
...
   <system.web>
   ...
    <authentication mode="Windows"/>
    <identity impersonate="true"/>
   ...
   </system.web>
...
</configuration>
```

Smart client applications automatically attach the Windows security context of the user currently logged on to the workstation. For this reason, this configuration step is not necessary if the client is a smart client application.

Service Setup

This task consists of the following three steps to configure the client computer to implement this pattern:

- **Enable SOAP extensions**. Includes steps to enable the service application to support the WSE 3.0 SOAP Protocol Factory.
- **Configure the policy**. Includes the WSE 3.0 policy configuration settings necessary to implement this pattern on the service.
- **Use the service code**. Includes the coding necessary to successfully implement this pattern on the service.

Enable SOAP Extensions

You must configure the service to enable SOAP extensions by performing the following steps.

► **To enable a Visual Studio 2005 project to support SOAP extensions**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable Microsoft Web Services Enhancement SOAP Protocol Factory** checkbox, and then click **OK**.

Configure the Policy

After enabling the service application to support WSE 3.0, you must enable policy support. If your application does not currently have a policy cache file, you can add one, and then perform the following procedure to enable policy support.

► **To add policy support to a WSE 3.0-enabled Visual Studio 2005 project**

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **Policy** tab, select the **Enable Policy** check box. Selecting this check box adds wse3policyCache.config as the default name for the policy cache file.
3. Under **Edit Application Policy**, click **Add**.
4. Type a policy friendly name for the new application policy, such as "KerberosService."

   Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides you with the choice to secure a service or a client computer. Select the option to **secure a service application** to configure the client computer.
6. The wizard also provides you with a choice of authentication methods in the same step. Select the authentication method for **Windows**, and then click **Next**.

7. On the **Kerberos Token Claims** page, the wizard presents configuration authorization based on the user name or roles contained in the **KerberosToken**. By default, the **perform authorization** check box is cleared. If you want to perform authorization through the policy assertion, select the **perform authorization** check box, and then add users and roles as appropriate.
8. On the **Message Protection** page, the wizard presents configuration options for message protection. Microsoft recommends selecting the option for **Sign**, **Encrypt**, **Encrypt Signature**. By default, the **Enable WS-Security 1.1 Extensions** check box is selected. You must enable the extensions if you want WSE 3.0 to use signature confirmation to correlate a response message returned from the service with the request message that prompted it.
9. If your primary concern is interoperability, clear the **Enable WS-Security 1.1 Extensions** check box.

   – or –

   If you do want to use signature confirmation, leave the **Enable WS-Security 1.1 Extensions** check box selected, click **Advanced Settings**, ensure that the **Enable signature confirmation** check box also is selected, and then click **OK**. The message protection settings must be the same for both the client computer and the service.
10. On the **Create Security Settings** page, click **Next**, review your settings, and then click **Finish**.

After you complete the procedure, your service security policy should appear similar to the following code example.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="KerberosService">
    <authorization>
      <allow role="Users" />
      <deny role="*" />
    </authorization>
    <kerberosSecurity establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="true" ttlInSeconds="300">
      <protection>
        <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <fault signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="false" />
      </protection>
    </kerberosSecurity>
    <requireActionHeader />
  </policy>
</policies>
```

The service's policy file is similar to the client's policy file with the following exceptions:

- An **<authorization>** assertion is included to limit which clients are allowed or denied access to the service by specifying a comma-separated list of roles that the user belongs to that are defined in the Active Directory domain. The **<allow>** element should appear first, with a list of authorized roles. If only specific roles are authorized to access the service, the **<deny>** element should always be specified immediately following the **<allow>** element with an asterisk (*). In the previous policy file, the **<authorization>** assertion is configured to allow only users who belong to the "Users" role, and to deny all others.
- The **<token>** section is not included, as the service does not attach a **KerberosToken** to the request message when calling another service.

The service also requires an additional configuration update to support secure conversations. The service's policy file contains two attributes in the **<kerberosSecurity>** element named **establishSecurityContext** and **renewExpiredSecurityContext** that are used to enable secure conversation. By default, both of these attributes are set to **true**. However, to support secure conversation, the service must disable stateful security context tokens (SCTs). This is accomplished by adding the following configuration entry to the Web.config file of the service host.

```
<microsoft.web.services3>
   <tokenIssuer>
      <statefulSecurityContextToken enabled="false" />
   </tokenIssuer>
</microsoft.web.services3>
```

Use the Service Code

This step describes the code required to implement the service. The following code example displays how the service is implemented. You can copy the code to insert it into a new Web service class file.

```
using System;
using System.Web.Services;

using Microsoft.Web.Services3;

using Microsoft.Practices.WSSP.WSE3.QuickStart.Common;

namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.MessageLayerKerberos.SecondService
{
    /// <summary>
    /// This class represents a web service used to query products catalog
    /// </summary>
    [WebService(Namespace =
"http://schemas.microsoft.com/WSSP/WSE3/QuickStart/BrokeredAuthentication/2005-
10/MessageLayerKerberos/SecondService.wsdl")]
    public class Service : System.Web.Services.WebService
    {
        /// <summary>
        /// Constructor
        /// </summary>
        public Service()
        {
            InitializeComponent();
        }

        #region Component Designer generated code

        //Required by the Web Services Designer
        private System.ComponentModel.Container components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
        }
```

(continued)

(continued)

```
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if(disposing && components != null)
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #endregion

        /// <summary>
        /// Returns some information about the specified product
        /// </summary>
        /// <param name="productName"></param>
        /// <returns></returns>
        [WebMethod]
        public Product GetProductInformation( string productName )
        {
            Product product = new Product();
            product = new Product();
            product.Name = productName;
            product.Quantity = 15;
            product.UnitPrice = 3.4M;
            return product;
        }
    }
}
```

In this code example, the service calls a second service that uses Windows Integrated Security. The service impersonates the client based on the received Kerberos token, delegated from the client to access the second Web service on behalf of the client. For more information about delegation, see Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Implementing Message Layer Security with Kerberos in WSE 3.0 pattern include the following:

- It provides single single-on capabilities that require a user to authenticate only once per session.
- It has broad acceptance as a brokered authentication protocol and the majority of large organizations that have centralized their authentication management infrastructure use the protocol.
- It is closely integrated with Windows (Windows 2000 or later). This enables the operating system to provide security capabilities such as impersonation, delegation, authorization, and auditing of the client.
- The Kerberos authentication process is more efficient than challenge/response. With the Kerberos protocol, authentication is performed by examining the security token sent in a request message. Challenge/response requires direct access to the authentication broker to authenticate a client.
- It supports mutual authentication when SPNs request a service ticket.
- It supports both signing and encryption of data in a Web service message.

Liabilities

The liabilities associated with the Implementing Message Layer Security with Kerberos in WSE 3.0 pattern include the following:

- The centralized nature of the Kerberos protocol requires a KDC to act as an authentication broker at all times. If the KDC fails, clients cannot establish new trust relationships with a service. Consider using redundant KDCs or providing an alternative mechanism, such as X.509 certificates for authentication. With Active Directory, you can improve KDC availability by establishing secondary domain controllers. This creates a redundant set of KDCs for the protocol to use.
- It is only useful for authentication and secure communication. In other words, the Kerberos protocol is not useful for securely persisting messages on a long-term basis because of the limited lifespan of tickets and session keys that it uses for encryption and signing.
- Proof that a client has authenticated cannot be established outside of the security domain where the client was authenticated, unless trust is explicitly established with the other security domain attempting to verify the client's security token.
- If you do not use signature confirmation, applications that use the **kerberosSecurity** policy are interoperable with applications implemented with the WS-Security 1.0 specification. If you do use signature confirmation, the application must support WS-Security 1.1.

- Implementing message layer security is likely to reduce the throughput and increase the latency of Web services, due to the overhead of the cryptographic operations that support canonicalization, XML signatures, and encryption. As part of your development process, you should identify performance objectives for your application and test the application against those objectives. For more information see, *Improving .NET Performance and Scalability*.

Security Considerations

Security considerations associated with the Implementing Message Layer Security with Kerberos in WSE 3.0 pattern include the following:

- When using Kerberos tokens at the message layer with Web services hosted on Windows 2000, the ASP.NET worker process requires higher privileges than it would normally possess.
- "Password guessing" attacks can occur against messages encrypted with a password equivalent (hash) that is derived from the user's password. The Kerberos protocol uses this derived key to encrypt data in the authentication request. An attacker could mount an offline dictionary attack by repeatedly attempting to decrypt the data in the authentication request sent to the KDC to discover the client's password.
- The Kerberos protocol does not implement authorization, although it is typically coupled with an authentication service that may store authorization information for a client. Resources can control access based on the client's authorization information, which is contained in the service ticket.

Note: Active Directory provides authorization services that complement its Kerberos implementation.

- You cannot use the Kerberos protocol to facilitate nonrepudiation, because the client's identity secret is shared with the KDC.

Implementing Message Layer Security with X.509 Certificates in WSE 3.0

Context

You are implementing brokered authentication in an application deployed on a Windows platform with security implemented at the message layer. A Web service using Web Services Enhancements (WSE) 3.0 is processing requests from clients. Clients and services use X.509 certificates with a standards-based security token that is portable across organizations and security boundaries. The solution must be able to provide a comprehensive set of security features that includes mutual authentication, data origin authentication, and data confidentiality.

Objectives

The objectives of this pattern are to:

- Secure a message exchange between two parties using brokered authentication with X.509 certificates.
- Combine an implementation of mutual authentication with the Data Origin Authentication and Data Confidentiality design patterns to provide a baseline that you can use to add more security requirements, such as replay protection and message validation.
- Demonstrate the implementation of the WSE 3.0 **mutualCertificate10Security** policy assertion and discuss when to use the **mutualCertificate11Security** policy assertion.
- Demonstrate an implementation of a custom WSE 3.0 **X509SecurityTokenManager** that allows you to associate additional data, such as roles with a client certificate.

Content

This pattern consists of the following sections:

- **Implementation Strategy**. This section provides a high-level description of the strategy to implement brokered authentication using X.509 certificates.
- **Implementation Approach**. This section describes the steps required to implement this pattern:
  - General setup
  - Configure the client
  - Configure the service

- **Resulting Context**. This section outlines the benefits, liabilities, and security considerations related to this pattern.
- **Extensions**. This section discusses how to extend the base pattern to implement role-based authorization.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

Use the **mutualCertificate10Security** policy assertion in WSE 3.0 to enable message signing and encryption using X.509 certificates. WSE 3.0 policy accesses the client's private key, which is used to sign the message. The service's public key, which is in its X.509 certificate, then encrypts the message. The service decrypts the message using its private key and verifies the signature using the public key of the client. The public key is in the client's X.509 certificate, which is included with the message.

Unlike using Secure Sockets Layer (SSL) in which the client obtains the service's X.509 certificate at run time, message layer security using X.509 certificates in WSE 3.0 requires that the service's certificate is obtained out-of-band and installed in the client's local certificate store.

Note: This pattern uses the **mutualCertificate10Security** policy assertion, because it relies on WS-Security 1.0. However, if your environment fully supports WS-Security 1.1 extensions, you can use the **mutualCertificate11Security** policy assertion. The **mutualCertificate11Security** policy assertion provides better performance, because it performs less asymmetric cryptography operations, which are computationally intensive. It performs two asymmetric and two symmetric operations compared with four asymmetric operations for the **mutalCertificate10Security** policy assertion.

This pattern assumes that the client has already obtained the service's certificate out-of-band, so that it can access the service's certificate from a local certificate store. For more information about installing X.509 certificates in the local certificate store, see How to: Use the X.509 Certificate Management Tools.

Participants

Using message layer security with X.509 certificates in WSE 3.0 involves the following participants:

- **Client**. The client accesses the Web service, and provides credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of the client to make access control decisions.

Process

The "Process" section of Brokered Authentication: X.509 PKI in Chapter 1, "Authentication Patterns," describes how you can use a certificate for authentication. This pattern provides a more detailed description of that process within the context of the implementation. The steps provided are based on the behavior of the **mutualCertificate10Security** assertion. The steps are divided into the following two sections based on what happens on the client and then on the service:

- The client initializes and sends a message with X.509 certificate information.
- The service authenticates the client using the X.509 certificate and signature.

The Client Initializes and Sends a Message with X.509 Certificate Information

This part of the process has six steps:

1. The client retrieves the service's X.509 certificate.
2. The client retrieves its own certificate and private key.
3. The client attaches its X.509 certificate to a message.
4. The client signs the message using its private key.
5. The client encrypts the message using the service's public key.
6. The client sends the message to the service.

The steps are summarized in Figure 3.8.

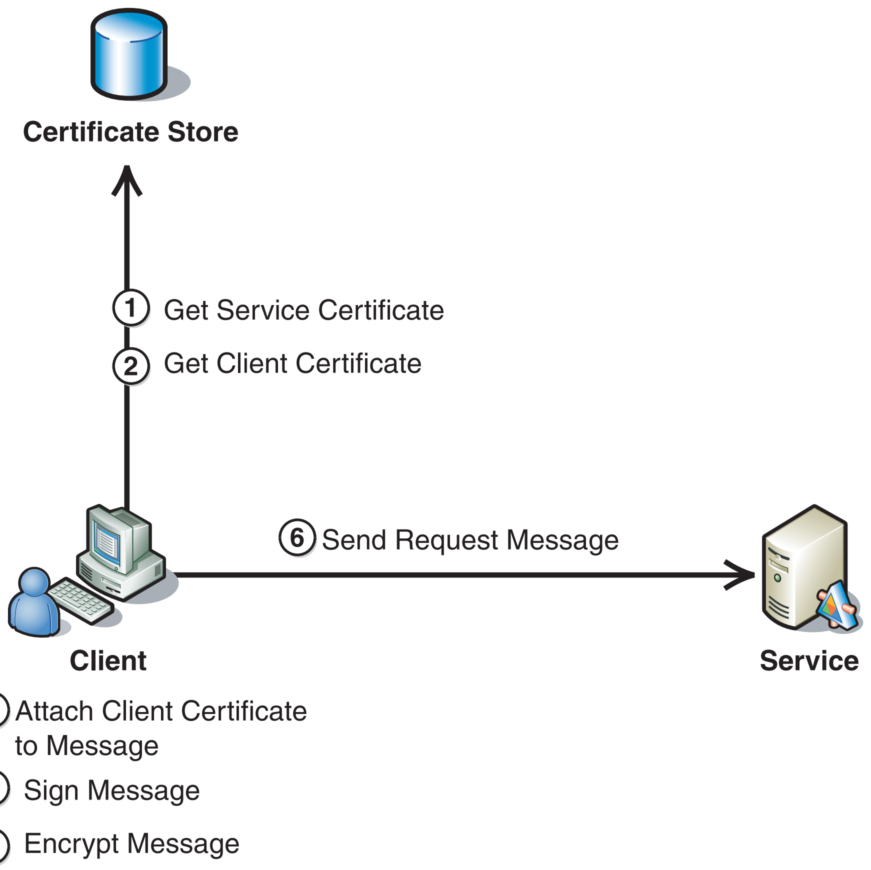


Figure 3.8
Initializing and sending a message with X.509 certificate information

Step One: The Client Retrieves the Service's Certificate

The client needs to access the X.509 certificate of the service to encrypt the request message. The WSE 3.0 policy assertion on the client is configured to retrieve the service's certificate from the client's local certificate store without the need for any additional code.

Step Two: The Client Retrieves Its own X.509 certificate and Private Key

The client accesses its X.509 certificate and private key. It uses the private key to sign the message and the X.509 certificate to provide the service with the public key and other information about the client for verification with the service.

Step Three: The Client Attaches Its X.509 Certificate to a Message

WSE 3.0 policy is configured to sign the message, and WSE 3.0 automatically attaches the client's certificate to the request message.

Step Four: The Client Signs the Message Using Its Private Key

The client uses its private key to sign the message. You can choose to sign one or more portions of the message, such as the address header or the message body. At a minimum, you should sign the message body, security, and addressing headers. A signature is created using a signature algorithm that computes a checksum value from the data to be signed and then encrypts the checksum value with the client's private key. When the signature is validated, the data used to create the signature is also validated to provide data origin authentication.

Step Five: The Client Encrypts the Message Using the Service's Public Key

You can encrypt message parts using a symmetric key that is encrypted with the public key from the service's X.509 certificate. At a minimum, ensure that the signature used to sign the encrypted data is itself encrypted to help protect it against offline attacks.

When you use WSE 3.0 policy to encrypt message data with X.509 certificates, the policy uses asymmetric encryption to encrypt a one-time symmetric key, which in turn encrypts the data. When message data is encrypted using the service's certificate information, WSE 3.0 also adds the certificate identifier to the message. If the certificate contains a subject key identifier, this is included to identify the certificate in the message. Otherwise, the policy uses the issuer name and certificate serial number instead. The service owns the certificate, which contains all the necessary information for it to access the appropriate private key and decrypt the symmetric key, which is then in turn used to decrypt the message.

Encrypting the request in this way protects sensitive data if the client is deceived into calling an illegitimate service. As the intended message recipient, only the correct Web service can decrypt the message with its private key.

Step Six: The Client Sends the Message to the Service

After the message is signed and encrypted, the client sends it to the service.

The Service Authenticates a Client Using the X.509 Certificate and Signature

This part of the process has six steps:

1. The service validates the client's certificate.
2. The service verifies the certificate trust chain.
3. The service checks the certificate revocation status.
4. The service decrypts the message.
5. The service verifies the signature.
6. The service initializes and sends a response to the client (optional).

The steps are summarized in the Figure 3.9.

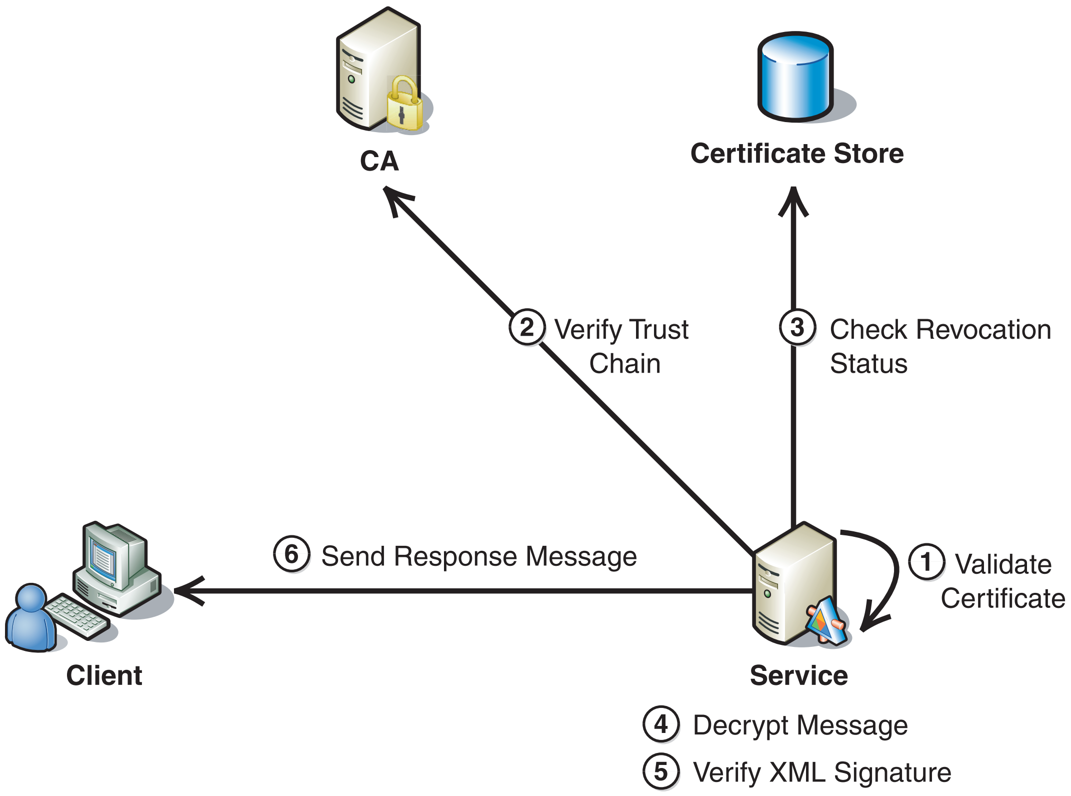


Figure 3.9
Authenticating a client using an X.509 certificate and signature

Step One: The Service Validates the Client's Certificate

WSE 3.0 validates the client's certificate attached to the request message. The certificate's validity period is checked to ensure that the service does not process a request that was secured with an expired X.509 certificate.

WSE 3.0 also verifies the integrity of the certificate's contents to ensure that it has not been tampered with after the certificate authority (CA) issued it. The integrity of the certificate's contents is verified using the signature of the issuing CA, which is also included in the certificate. If the certificate's contents cannot be validated against the issuer's signature, then the certificate has been tampered with and it is rejected as invalid. For more information about the contents of an X.509 certificate, see the X.509 Technical Supplement in Chapter 7, "Technical Supplements."

Step Two: The Service Verifies the Certificate Trust Chain

By default, WSE 3.0 verifies the trust chain of certificates, or requires that the client's certificate is installed in the **Trusted People** folder in the service's local certificate store. WSE 3.0 must be able to recognize an issuing CA as trusted to verify the certificate trust chain for the client's X.509 certificate. WSE 3.0 recognizes an issuing CA as trusted based on the X.509 certificate that endorses the client's certificate. WSE 3.0 recognizes the issuing CA's certificate as a trusted root for a certificate chain if the CA's X.509 certificate is installed in the machine certificate store in the **Trusted Root Certification Authorities** folder.

The high-level steps to install a certificate chain are as follows:

1. Export the certificate chain from the CA. This is dependant on the type of CA that issued the certificate.
2. Import the certificate chain into a local certificate store.

Note: For more information about managing certificates and trust chains, see the X.509 Technical Supplement in Chapter 7, "Technical Supplements."

Step Three: The Service Checks the Certificate Revocation Status

WSE 3.0 policy checks the revocation status of the certificate by verifying whether the certificate is on a certificate revocation list (CRL) that the CA publishes. You can obtain the CRL out-of-band by downloading it from a CA, and then importing it into a local certificate store where WSE 3.0 can access it. You can also check the revocation status of the certificate online. However, this approach relies on an online revocation service that the service must access to verify the certificate's revocation status. There is also a performance cost associated with checking the revocation status online. For this reason, you may want to consider downloading the CRL instead, if you can frequently update the cached CRL. By default, WSE 3.0 verifies the revocation status of X.509 certificates online.

Note: For more information about CRLs, see the X.509 Technical Supplement in Chapter 7, "Technical Supplements."

Step Four: The Service Decrypts the Message

By default, the **mutualCertificate10Security** assertion protects the message body by encrypting it. When WSE 3.0 receives an encrypted message, WSE 3.0 policy automatically decrypts it using the following steps:

1. WSE determines the value to identify the service's certificate — either the RFC3280 Subject Key Identifier, or the issuer name and serial number — that the client included in the message tells the service which certificate was used to encrypt the message. WSE 3.0 policy uses this value to determine which private key it must use to decrypt the message.
2. WSE decrypts the asymmetrically encrypted, one-time symmetric key that the client sent with the message, using the service's private key
3. WSE uses the symmetric key to decrypt the message data using a symmetric algorithm. By default, WSE 3.0 uses AES 256 for symmetric encryption.

Note: Service side policy alone does not stop a client from sending an unencrypted message. However, policy will reject a message at the server if it is not encrypted.

Step Five: The Service Verifies the Signature

WSE 3.0 verifies the client's signature on the incoming request message using the public key sent with the message. If the message data is signed, this step also validates the client as the message originator to provide data origin authentication.

Step Six: The Service Initializes and Sends a Response to the Client (Optional)

If the service returns a secure response to the client, the same process described in these steps is used for the response message between the service and the client, except that the roles of the client and the service reverse. However, unlike the request message, the service does not attach its X.509 certificate to the response message, because the client already has a copy of it.

Instead, WSE 3.0 policy adds a reference to the service's certificate in the response message. The service initiates and sends the response, signs it with the service's private key and encrypts it with a symmetric key that is encrypted with the client's X.509 certificate public key. The client processes the response in the same manner as the service processed the request: decrypt the symmetric key with the client's private key, and then decrypt the encrypted message parts with the symmetric key. Finally, the client verifies the service's signature with the service's X.509 certificate.

Implementation Approach

This section describes how to implement this pattern. The section is divided into three major tasks:

1. **General setup**. This task provides the required steps for both the client and the service.
2. **Configure the client**. This task provides the required steps to configure WSE 3.0 policy and the code on the client.
3. **Configure the service**. This task provides the required steps to configure WSE 3.0 policy and the code on the service.

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

General Setup

You must install WSE 3.0 on the computers that you use to develop WSE-enabled applications. After WSE 3.0 is installed, you must enable the client and the service to support WSE 3.0. You can achieve this by performing the following steps:

► **To enable a Visual Studio 2005 project to support WSE 3.0**

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable this project for Web Services Enhancements** check box, and then click **OK**.

Both the client and service require access to their respective X.509 certificates from the local certificate stores on the host computers. The client also requires access to its private key, and the service requires access to its private key. Also, the client must be able to access the service's X.509 certificate from its local certificate store. Typically, the certificate for a trusted service is installed in the **Trusted People** folder in the local certificate store. For more information about how to install X.509 certificates in the local machine certificate store, see the Certificates How To.

Note: You can use the WSE Certificates tool to view private key file properties and set access permissions for the account under which the client and service run.

For applications that use X.509 certificates, X.509 security must be configured for WSE 3.0.

► To configure WSE 3.0 X.509 security settings

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. Click the **Security** tab, and then select the **allow test roots** check box if you are using a self-signed test certificate in a development or test environment. If you are configuring the application to run in a production environment, leave this check box cleared.
3. For **Revocation mode**, select the **Offline** option if you do not want to depend on accessing the certificate's revocation status online. If you select this option, you must be confident that you can update a local copy of the CRL in the local certificate store frequently enough to meet your requirements for certificate verification. If you want to allow the application to access the revocation status online, leave this option set at the default value **Online**, and then click **OK**.

After configuring the settings for X.509 certificate security with the WSE 3.0 Settings tool, they should appear in the application configuration file, as shown in the following XML example.

```
<configuration>
...
   <microsoft.web.services3>
      <security>
         <x509 verifyTrust="true" allowTestRoot="true" revocationMode="Offline"
verificationMode="TrustedPeopleOrChain"/>
      </security>
...
    </microsoft.web.services3>
...
</configuration>
```

Note: Usually, it is not necessary to modify this information directly because you can control these settings through the WSE 3.0 Settings tool.

The **allowTestRoot** attribute shown in the previous example determines whether the application allows test certificates. Test certificates are acceptable for development and test environments. However, for production environments you should only use certificates issued by a CA. This attribute is optional, and its value is **false** by default.

If you set the **verificationMode** attribute to **TrustedPeopleOrChain** to verify the signature on an incoming message, this setting requires that the message sender's X.509 certificate is located in the **Trusted People** folder of the verifying party's certificate store or that the certificate can be verified to a trusted CA through a certificate trust chain. For more information about configuring the behavior of X.509 security in WSE 3.0, see <x509> Element on MSDN.

Note: WSE 3.0 offers four different protection levels that determine how messages are secured using SOAP message security. Generally, you should use the **Sign, Encrypt, and Encrypt Signature** setting for best message protection. This setting encrypts the message body and the XML signature, which reduces the likelihood of a successful cryptographic guessing attack against the signature. For this reason, all the composite implementation patterns use this value as default. If you want to use this setting in new Web services you should change the **messageProtectionOrder** attribute to the following value in your security policy:

```
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
```

Configure the Client

After enabling the client application to support WSE 3.0 during General Setup, you must enable policy support. If your application does not currently have a policy cache file, you can add one for this purpose, and enable policy support by performing the following steps.

► **To add policy support to a WSE 3.0-enabled Visual Studio 2005 project**

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **Policy** tab, select the **Enable Policy** check box. Selecting this setting adds a policy cache file with the default name wse3policyCache.config.
3. Under **Edit Application Policy**, click **Add**, and then type a policy friendly name for the new application policy, such as "x509."
4. Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides a choice to secure a service or a client. Select **secure a client application** to configure the client. The wizard also provides a choice of authentication methods in the same step. Select **Certificate**, and then click **Next**.
6. On the **Client Certificate** page, select the client certificate for the client. Unless your client application is impersonating a Windows user, select **LocalMachine** for the Store Location.
7. Click **Select Certificate** to select the appropriate X.509 certificate for the client application, click **OK**, and then click **Next**.
8. On the **Message Protection** page, the wizard displays configuration options for message protection. By default, the **Enable WS-Security 1.1 Extensions** check box is selected. Clear this check box to use the **mutualCertificate10Security** assertion. Leave it selected if you want to use the **mutualCertificate11Security** assertion. For **Protection Order**, select **Sign, Encrypt, Encrypt Signature**, and then click **Next**.
9. On the **Server Certificate** page, select **LocalMachine** for the Store Location, click **Select Certificate**, select the appropriate X.509 certificate for the service, click **OK**, and then click **Next**.
10. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete these tasks, your client security policy should look similar to the following code example.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="x509">
    <mutualCertificate10Security establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="false" ttlInSeconds="300">
      <clientToken>
        <!-- WSE2 QuickStart Client Certificate -->
        <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartClient" findType=" FindBySubjectDistinguishedName"/>
      </clientToken>
      <serviceToken>
        <!-- WSE2 QuickStart Server Certificate -->
        <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartServer" findType="FindBySubjectDistinguishedName" />
      </serviceToken>
      <protection>
        <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <fault signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="false" />
      </protection>
    </mutualCertificate10Security>
    <requireActionHeader/>
  </policy>
</policies>
```

In the previous policy example, the **signatureConfirmation** attribute is set to **false**. If this value is set to **true**, compatibility with WS-Security 1.0 is lost.

The **<clientToken>** element contains values that specify the client's X.509 certificate for signing outbound request messages and decrypting inbound response messages from the service. The information specific to the X.509 certificate is contained in the **<x509>** element. Set the **findValue** attribute to the value used to locate the certificate within the local certificate store. This will be the subject distinguished name for certificates retrieved from the certificate store using the client name or the text encoded binary value for a certificate identifier, such as the certificate's SHA1 thumbprint.

If you use the WSE 3.0 default configuration that retrieves certificates from the certificate store according to the subject distinguished name, you may risk confusing the identity of the client if different CAs issue different certificates with the same subject distinguished name. To avoid this, consider using a certificate identifier to retrieve certificates from the certificate store. For more information about how to set the **findType** and **findValue** attributes for the **<x509>** element, see <x509> Element (Policy) in the WSE 3.0 documentation.

Note: The value that WSE 3.0 is configured to use to find the certificate in the certificate store is not directly connected to the identifier that WSE 3.0 uses to identify certificates in transit when messages are sent. For example, by default, WSE 3.0 retrieves certificates from the certificate store by subject distinguished name, but the **mutualCertificate10Security** assertion uses either the RFC3280 subject key identifier or the issuer name and serial to identify the certificate in the request message.

In this example, the certificate and corresponding private key that the client uses belong to an application, not a user. If you want to authenticate a user instead of a client application, use a smart client application and ensure that the user's certificate and private key are accessible from the local certificate store on the workstation where the smart client application is installed. To ensure that the private key and corresponding certificate are only accessible to the user of the smart client application, set the **Store Location** to **CurrentUser** in step 6 of the previous procedure when configuring security policy on the client. Also, set the access control list (ACL) on the private key file so that only the user that owns the certificate can access it.

You can use the WSE 3.0 Certificates tool to obtain certificate information, such as the subject distinguished name or subject key identifier. The WSE 3.0 Certificates tool displays the subject distinguished name in reverse order, but this is actually the correct order for the client name in a WSE 3.0 policy assertion. You can also use the Microsoft Management Console (MMC) snap-in to obtain the subject distinguished name or certificate thumbprint, but as the subject distinguished name is not reversed, you must reverse it yourself to use it in a WSE 3.0 policy assertion. For example, you must reverse a subject distinguished name obtained from the MMC snap-in such as "CN=bob, DC=Microsoft, DC=com" when you specify it in policy to read as "DC=com, DC=Microsoft, CN=bob."

If you configure your application to retrieve certificates from the certificate store using the certificate's SHA1 thumbprint by setting the **findType** value of the **<x509>** attribute to **FindByThumbprint**, WSE 3.0 requires you to set the **findValue** attribute to the hexadecimal encoded value for the certificate thumbprint, not the Base64 encoded value. You must use the Certificates MMC snap-in to obtain this value, and remove the spaces between each byte value. For example, the first few bytes of a certificate thumbprint copied from the Certificates MMC snap-in would be formatted as: "c6 74 47 da..." Remove the spaces when pasting this information into the **findValue** attribute so that it displays as: "c67447da..." You cannot obtain this value using the WSE 3.0 Certificates tool.

The **<serviceToken>** element contains information about the service's certificate, which encrypts request messages and verifies the signature on response messages. The **<serviceToken>** settings can be configured similarly to those of the **<clientToken>** described previously in this section.

For more information about configuring other settings for this policy assertion, see <mutualCertificate10> Element in the WSE 3.0 documentation.

When you add a Web reference to the service from the client application to create a Web service proxy, two proxies are generated for the Web service: one is a nonWSE 3.0 proxy and the other is WSE 3.0–enabled. This pattern uses the WSE 3.0-enabled proxy class, which is name + “Wse.” For example, if your Web service is named “MyService,” your WSE 3.0–enabled Web service proxy class name would be “MyServiceWse.”

The following code example demonstrates how to bind the policy assertion described previously for calling the Web service proxy. This example uses a WSE 3.0–enabled Web service proxy.

```
...
using System.Globalization;
...
try
{
   Service.ServiceWse service = new Service.ServiceWse();

   service.SetPolicy("x509");

   Service.Product product = service.GetProductInformation(txtProductName.Text);

   txtResults.Text = String.Format(CultureInfo.InvariantCulture,
                    "Product Name: {0}, Unit Prize: {1}, Quantity: {2}",
                    product.Name, product.UnitPrice, product.Quantity);

}
catch (Exception ex)
{
   txtResults.Text = ex.ToString();
}
...
```

Configure the Service

You must configure the service to enable SOAP extensions by performing the following steps.

► **To enable a Visual Studio 2005 project to support SOAP extensions**

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable Microsoft Web Services Enhancement SOAP Protocol Factory** check box, and then click **OK**.

After you enable the service to support WSE 3.0, you also must enable policy support. If your application does not currently have a policy cache file, you can add one, and enable policy support by performing the following steps.

► To add policy support to a WSE 3.0-enabled Visual Studio 2005 project

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0.**
2. On the **Policy** tab, select the **Enable Policy** check box. Selecting this setting adds a policy cache file with the default name wse3policyCache.config.
3. Under **Edit Application Policy**, click **Add**, and then type a policy friendly name for the new application policy, such as "x509."
4. Click **OK** to start the WSE 3.0 Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides a choice to secure a service or a client. Select **secure a service application** to configure the service. The wizard also provides a choice of authentication methods. Select **Certificate**, and then click **Next**.
6. On the **Authorized Clients** page, the wizard presents the option to perform authorization. If you want to add authorization to your service policy, select the **Perform Authorization** checkbox, click **Add** to add the X.509 certificates for the clients that you want to authorize to call the service, and then click **Next**.
7. On the **Message Protection** page, the wizard displays configuration options for message protection. By default, the **Enable WS-Security 1.1 Extensions** check box is selected. Clear this check box to use the **mutualCertificate10Security** assertion. Leave it selected if you want to use the **mutualCertificate11Security** assertion. For **Protection Order**, select the option for **Sign, Encrypt, Encrypt Signature**, and then click **Next**.

   Note: These settings must be the same on the client and the service.

8. On the **Server Certificate** page, click **Select Certificate** to select the appropriate X.509 certificate to use for the service, click **OK**, and then click **Next**.
9. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete these tasks, your service security policy should look similar to the following code example.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="x509">
    <authorization>
      <allow user="CN=WSE2QuickStartClient" />
      <deny user="*" />
    </authorization>
    <mutualCertificate10Security establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="false" ttlInSeconds="300">
```

(continued)

(continued)

```
      <serviceToken>
        <!-- WSE2 QuickStart Server Certificate -->
        <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartServer" findType="FindBySubjectDistinguishedName" />
      </serviceToken>
      <protection>
        <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
        <fault signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="false" />
      </protection>
    </mutualCertificate10Security>
    <requireActionHeader />
  </policy>
</policies>
```

The service's policy file is similar to the client's policy file with the following exceptions:

- An **<authorization>** assertion limits which clients are allowed to access the service by specifying a comma-separated list of client names as they appear in the X.509 certificate's subject name attribute. The **<allow>** element should appear first with a list of authorized clients. If only specific clients are authorized to access the service, the **<deny>** element should always be specified immediately after the **<allow>** element with an asterisk (*) in its **user** attribute. This prevents access by all clients except those using X.509 certificates that are explicitly authorized in the **<allow>** element.
- The **<clientToken>** element is not specified because the service uses the X.509 security token attached to the request message to verify the signature on the request message, and to encrypt the response for transmission to the client.

The following code example demonstrates how to apply the policy provided previously when the service processes a request.

```
using System;
using System.Web.Services;

using Microsoft.Web.Services3;

using Microsoft.Practices.WSSP.WSE3.QuickStart.Common;

namespace Microsoft.Practices.WSSP.WSE3.QuickStart.MessageLayerX509.Service
{
    ///<summary>
    ///This class represents a Web service used to query products catalog.
    ///</summary>
```

(continued)

(continued)

```
    [WebService(Namespace =
"http://schemas.microsoft.com/WSSP/WSE3/QuickStart/BrokeredAuthentication/2005-
10/MessageLayerX509.wsdl")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [Policy("x509")]
    public class Service : System.Web.Services.WebService
    {
        ///<summary>
        ///Constructor
        ///</summary>
        public Service()
        {
        }

        ///<summary>
        ///Returns some information about the specified product.
        ///</summary>
        ///<param name="productName"></param>
        ///<returns></returns>
        [WebMethod]
        public Product GetProductInformation( string productName )
        {
            Product product = new Product();
            product.Name = productName;
            product.Quantity = 10;
            product.UnitPrice = 2.5M;
            return product;
        }
    }
}
```

In the previous code example, a reference to **Microsoft.Practices.WSSP.WSE3.QuickStart.Common** found in the WSSP QuickStarts code provides a reference to the **Product** class. Replace these as necessary for your application.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Implementing Message Layer Security with X.509 Certificates in WSE 3.0 pattern include the following:

- Authentication, data confidentiality, and data origin authentication using a single security mechanism.
- Authentication can occur over well-known Internet firewall friendly ports using well-known protocols (for example, HTTP/HTTPS over port 80/443).
- Authentication and message protection can occur across organizational boundaries and security domains, because you do not need to propagate the private key.

Liabilities

The liabilities associated with using the Implementing Message Layer Security with X.509 Certificates in WSE 3.0 pattern include the following:

- You need to store the private keys somewhere securely, such as on a smart card or a computer, which makes them less portable than passwords.
- The use of asymmetric cryptography is computationally intensive and may cause performance issues, even though WSE 3.0 optimizes asymmetric cryptography for performance. Most of the time, you can mitigate this issue by deploying servers with more processors or by adding more servers to a load balancing cluster.
- Signature verification using test certificates generated using the MakeCert utility can cause serious performance issues. You can use certificates issued by a CA to mitigate this issue. For more information about obtaining certificates, see the X.509 Technical Supplement in Chapter 7, "Technical Supplements."
- Implementing message layer security is likely to reduce the throughput and increase the latency of Web services, due to the overhead of the cryptographic operations that support canonicalization, XML signatures, and encryption. As part of your development process, you should identify performance objectives for your application and test the application against those objectives. For more information, see *Improving .NET Performance and Scalability*.

Security Considerations

Security considerations associated with using the Implementing Message Layer Security with X.509 Certificates in WSE 3.0 pattern include the following:

- Web services are susceptible to man-in-the-middle attacks, where an attacker could replace the signature and certificate information. To mitigate such an attack, the service must be able to limit the population of potential clients to a trusted group, either individually based on the client's X.509 certificate or as a group through a limited population of clients that are defined by a certificate trust chain. You can specify an **<authorization>** assertion in the service policy to restrict the clients that are allowed to access the service based on their subject distinguished name.

- When using X.509 certificates for authorization, WSE 3.0 only allows authorization to occur based on a certificate's subject distinguished name, not by certificate identifier. This creates the potential for confusion if there are different certificates issued by different CAs with the same subject distinguished name. If both CAs are trusted by the service, WSE 3.0 cannot distinguish between the certificates for authorization purposes. For this reason, it is especially important to verify certificate trust chains. A service that does not require trust chain verification could be exploited by an attacker creating a bogus certificate with the same subject distinguished name as an authorized certificate, and then using it to access the service. A potential solution to this problem is to extend the **X509SecurityTokenManger** released with WSE 3.0 to return a certificate identifier, such as the certificate's SHA1 thumbprint, instead of the subject distinguished name for authorization checks. A certificate identifier provides a more distinct way to identify the certificate than a subject distinguished name. For more information about this subject, see the next section.
- WSE 3.0 does not encrypt the client certificate that is attached to a request message. If you need to protect the identity of clients from disclosure to eavesdroppers, this introduces a potential information disclosure vulnerability. This is because certificates often contain information that eavesdroppers can use to identify the client.

Extensions

This section provides examples of how to extend the base pattern to provide additional security features.

Role-based Authorization

The lifetime of an X.509 certificate is typically greater than that of other security token types. As a result, it is difficult to provide security roles directly in an X.509 certificate. This would require you to use extensibility mechanisms to provide custom role information in the certificate. Because the role memberships of the certificate owner are likely to change before the certificate expires, the CA would need to issue a new certificate every time the certificate owner's roles change.

It is possible to establish a security context for a client that has successfully authenticated using a X.509 certificate by associating roles with its X.509 certificate. You can accomplish this by implementing a custom **X509SecurityTokenManager** on the service to construct a security principal, and then attach it to the security token. You can retrieve the roles for the security principal from a database, Active Directory, or another service that can provide roles for an identity to eliminate the need to provide them directly within the certificate.

The following code example provides an example of a custom **X509SecurityTokenManager**. After the **CustomX509SecurityTokenManager** authenticates the client, it constructs a **GenericPrincipal** with security roles and attaches it to the security token. You can copy this code and paste it into a new class file. However, you must provide code where indicated by comments to retrieve user roles from a database or other service provider, and change the namespace to suit your project.

```
using System;
using System.Xml;
using System.Security.Cryptography.Xml;
using System.Security.Principal;

using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;

namespace Microsoft.Practices.WSSP.WSE3.QuickStart.MessageLayerX509.Service
{
    /// <summary>
    /// By implementing X509SecurityTokenManager we can manipulate the token
    /// on messages received.
    /// </summary>
    public class CustomX509SecurityTokenManager : X509SecurityTokenManager
    {
        /// <summary>
        /// Constructs an instance of this security token manager.
        /// </summary>
        public CustomX509SecurityTokenManager()
        {
        }

        /// <summary>
        /// Constructs an instance of this security token manager.
        /// </summary>
        /// <param name="nodes">An XmlNodeList containing XML elements from a
configuration file.</param>
        public CustomX509SecurityTokenManager(XmlNodeList nodes)
            : base(nodes)
        {
        }

        /// <summary>
        /// Adds a generic principal to the token
        /// </summary>
        /// <param name="token">The X509SecurityToken token</param>
        protected override void
AuthenticateToken(Microsoft.Web.Services3.Security.Tokens.X509SecurityToken token)
        {
            base.AuthenticateToken(token);

            // Assigns certificate's hexadecimally encoded SHA1 thumbprint to
GenericIdentity
            // Certificate's hexadecimally encoded SHA1 Thumbprint value can be
obtained using the Certificates MMC Snap-In
```

(continued)

(continued)

```
            string subjectKeyIdentifier = token.Certificate.Thumbprint;

            GenericIdentity identity = new GenericIdentity(subjectKeyIdentifier);
         //Replace the next line with your own code to retrieve roles from a role
store and populate the GenericPrincpal
            GenericPrincipal principal = new GenericPrincipal( identity, new
string[] {"role1, role2, role3"} );

            token.Principal = principal;
        }
    }
}
```

To use the previous example **CustomX509SecurityTokenManager**, you must create a security token manager entry in the service's Web.config file.

```
...
<microsoft.web.services3>
   ...
   <security>
   ...
      <binarySecurityTokenManager>
         <add type="
Microsoft.Practices.WSSP.WSE3.QuickStart.MessageLayerX509.Service.CustomX509Securi
tyTokenManager"
 valueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-
profile-1.0#X509v3"/>
      </binarySecurityTokenManager>
   </security
   ...
</microsoft.web.services3>
```

In the previous configuration example, you must modify the fully qualified class name for the custom security token manager to match the code for your **CustomX509SecurityTokenManager**.

With this extension to the base implementation, you can perform role-based authorization on the principal attached to the **X509SecurityToken** and avoid the limitation of authorization checks based solely on the identity represented in the certificate. You can perform authorization using policy or code. For an authorization example, see the service policy example under Configure the Service.

This extension also addresses the issue in the Security Considerations section of the base pattern about the ability to only perform identity-based authorization on the certificate's subject distinguished name, not based on a certificate identifier. While the default **X509SecurityTokenManager** adds the certificate's subject distinguished name to the **GenericIdentity**, the **CustomX509SecurityTokenManager** defined in this extension assigns the client certificate's hexadecimally encoded value of the SHA1 thumbprint to the security token identity instead of the certificate's subject distinguished name.

By assigning the client certificate's SHA1 thumbprint to the security token identity, you can use the hexadecimally encoded value of the client certificate's SHA1 thumbprint in the service's **<authorization>** assertion instead of the subject distinguished name. You can use the Certificates MMC Snap-In tool to obtain the hexadecimally encoded SHA1 thumbprint for a certificate. The following XML example provides an example of how to use the client certificate's SHA1 thumbprint in an **<authorization>** assertion in the service's policy cache.

```
...
<authorization>
      <allow user="ca7601381b4578502b62b8809825664f1e78dfa2" />
      <deny user="*" />
</authorization>
...
```

This code example mitigates the risk of confusing client identities by providing a way to identify client certificates that is more likely to be unique, as it performs authorization on the service when CAs issue different certificates with the same subject distinguished name.

Implementing Message Layer Security with a Security Token Service (STS) in WSE 3.0

Note: This pattern is currently under development. It is due for release in early 2006.

Context

You are implementing brokered authentication in an application deployed on computers running Windows operating system software with security implemented at the message layer. Web services need to authenticate clients in a heterogeneous environment so that you can implement additional controls, such as authorization and auditing. The authentication broker negotiates trust between client applications and Web services, which removes the need for a direct relationship. The authentication broker should issue signed security tokens for authentication.

Implementation Strategy

A QuickStart that demonstrates how to develop a Web Service Enhancements (WSE) 3.0 Security Token Service (STS) that issues XML tokens is currently under development. This pattern will be updated when the QuickStart is released.

If you are interested in obtaining a Community Technical Preview (CTP) release or would like to contribute requirements, join the Security Token Service Quickstart community workspace.

References for Transport Layer Security

There is a lot of good information available on using transport layer security to secure Web services, so this information is provided in the form of the following references, which point you to appropriate guidance for implementing transport layer security. It contains the following sections:

- Implementing Brokered Authentication Using Windows Integrated Security on IIS
- Implementing Transport Layer Data Confidentiality Using HTTPS
- Implementing Transport Layer Security Using HTTP Basic over HTTPS
- Implementing Transport Layer Security Using X.509 Certificates and HTTPS
- Implementing Transport Layer Security with Kerberos and IPSec on Windows Server 2003

Implementing Brokered Authentication Using Windows Integrated Security on IIS

This implementation reference provides guidance for implementing brokered authentication on an existing Kerberos version 5 protocol infrastructure at the transport layer. Brokered authentication using Windows Integrated Security on Internet Information Services (IIS) 6.0 allows you to call applications and Web services to validate credentials against an Active Directory domain controller, as an implementation of the Kerberos protocol. The calling applications and Web services can validate credentials against the same Active Directory domain or multiple Active Directory domains joined by a cross-domain trust relationship. The Web services also can impersonate the caller to access resources controlled under a trusted Active Directory domain.

To implement brokered authentication using Windows Integrated Security on IIS 6.0, you must perform the following tasks:

1. Implement transport-layer brokered authentication using Windows Integrated Security.
2. Configure IIS 6.0 to require Windows Integrated Security.
3. Add the credentials for the client that the Web service will authenticate to the credential cache of the Web service proxy that communicates with the Web service.

The benefits to this approach include:

- A minimal amount of code and configuration work for the implementation.
- When you implement this approach using Kerberos authentication instead of NTLM authentication, you can use it to flow the caller’s identity across multiple system hops.

One liability to this approach is that firewall boundaries may not allow Kerberos authentication traffic between the calling application and the Kerberos Key Distribution Center (KDC) or between the Web service and the Kerberos KDC.

It is also important to take into account that this approach does not provide data confidentiality or data origin authentication for messages sent between the calling application and the Web service. Use HHTPS or IPSec to secure messages between the calling application and Web service. For more information about these limitations, see, "Implementing Transport Layer Security Using HTTP Basic over HTTPS" and "Implementing Transport Layer Security Using Kerberos and IPSec on Windows Server 2003."

For more information about implementing this approach, see the following resources:

- To learn more about Windows Integrated Security, see the "Authentication and Authorization Strategies" section in "Web Services Security" on MSDN.
- To call a Web service configured to use Windows Integrated Authentication, see the "Passing Credentials for Authentication to Web Services" section in "Web Services Security" on MSDN.

Implementing Transport Layer Data Confidentiality Using HTTPS

This implementation reference provides guidance on implementing data confidentiality using X.509 certificates at the transport layer. To implement data confidentiality using X.509 certificates at the transport layer, you must perform the following tasks:

1. Implement transport layer security using Secure Sockets Layer (SSL).
2. Configure the Web service virtual directory in IIS 6.0 to require SSL.

One benefit to this approach is that SSL is a well-established protocol that is easy to configure and implement on the Windows platform. However, there are several liabilities and security considerations to take into account with this approach. You can only establish SSL point-to-point as opposed to end-to-end, as message layer security is capable of doing. Additional liabilities of this approach include:

- Communication between several points configured for SSL rather than end-to-end at the message layer may cause unacceptable application response times.
- All points in the communication must be sufficiently trusted to establish SSL.

In some cases, these liabilities may warrant using a different approach, such as implementing message layer X.509 security using X509Security Tokens in WSE 3.0.

This approach has the following security considerations:

- Vulnerabilities exist in previous versions of SSL. The server and client need to have security patches installed on them to mitigate known vulnerabilities.
- Microsoft strongly recommends configuring IIS 6.0 to require strong (128-bit) SSL encryption for increased protection of data confidentiality.

For more information about implementing this approach, see the following resources:

- To learn how an SSL session is established between two parties, see "Description of the Secure Sockets Layer (SSL) Handshake" on Microsoft Help and Support: *http://support.microsoft.com/default.aspx?scid=kb;%5bLN%5d;Q257591*.
- To learn how to implement SSL, see:
  - "How To Set Up SSL on a Web Server" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod30.asp*.
  - "How To Call a Web Service Using SSL" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod28.asp*.
  - "How To Call a Web Service Using Client Certificates from ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod27.asp*.

Implementing Transport Layer Security Using HTTP Basic over HTTPS

This implementation reference provides guidance on implementing direct authentication using HTTP Basic over HTTPS on the transport layer. An advantage of this implementation is that it can make use of an existing infrastructure.

To implement transport layer security using HTTP Basic over HTTPS, you must perform the following tasks:

1. Implement transport layer direct authentication using HTTP basic authentication.
2. Configure IIS 6.0 to require HTTP basic authentication for the virtual directory hosting the service.
3. On the client, add the client's credentials to the credential cache of the proxy that communicates with the service.

This approach is generally considered easy to configure and simple to use. It uses a well established and widely supported type of direct authentication in a Web environment. However, one liability to this approach is that it provides no message protection capabilities. Using HTTP basic authentication, the client's credentials are passed in plaintext in transit, which makes them easily susceptible to eavesdropping by an attacker. Therefore, Microsoft strongly recommends using SSL to provide data confidentiality to prevent eavesdropping attacks against the credentials.

For more information about implementing this approach, see the following resources:

- To learn how to configure IIS for HTTP basic authentication, see "Basic Authentication in IIS 6.0" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/abbca505-6f63-4267-aac1-1ea89d861eb4.mspx*.
- To learn how to call a Web service that requires credentials, see the "Passing Credentials for Authentication to Web Services" section in "Web Services Security" on MSDN: *http://msdn.microsoft.com/library/en-us/secmod/html/secmod10.asp*.

Implementing Transport Layer Security Using X.509 Certificates and HTTPS

This implementation reference provides guidance for implementing brokered authentication using X.509 certificates on the transport layer. Transport layer security using X.509 certificates and HTTPS secures point-to-point communication. Messages do not require intermediaries to process them and they are not securely persisted for any period of time.

To implement transport layer security using X.509 certificates and HTTPS, you must perform the following tasks:

1. Implement transport layer security using SSL.
2. Configure the Web service virtual directory to use SSL and require client certificates.

This approach has the following benefits:

- It provides brokered authentication, data confidentiality, and data origin authentication capabilities in one solution.
- It uses SSL, which is a well established protocol that is easy to configure and implement on the Windows platform.

The disadvantage of this approach is that you can only establish point-to-point SSL, not end-to-end as message layer security is capable of doing. There are certain liabilities as a result of using SSL that may warrant you to use a different approach, such as implementing message layer X.509 security using X509Security Tokens with WSE 3.0. These liabilities include:

- Communication between several points configured for SSL rather than end-to-end at the message layer may cause unacceptable application response times.
- All points in the communication must be sufficiently trusted to establish SSL.

This approach has the following security considerations:

- Vulnerabilities exist in previous versions of SSL. The server and client need to have security patches installed to mitigate known vulnerabilities.
- Microsoft strongly recommends configuring IIS 6.0 to require strong (128-bit) SSL encryption for increased protection of data confidentiality.

For more information about implementing this strategy, see the following resources:

- To learn about how an SSL session is established between two parties, see "Description of the Secure Sockets Layer (SSL)" on Microsoft Help and Support: *http://support.microsoft.com/default.aspx?scid=kb;%5bLN%5d;Q257591*.
- To learn about how a client authenticating to a service using SSL operates, see "Description of the Client Authentication Process During the SSL Handshake" on Microsoft Help and Support: *http://support.microsoft.com/kb/257586/EN-US/*.
- To learn about how to implement SSL, see the following documentation:
  - "How To Set Up SSL on a Web Server" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod30.asp*.
  - "How To Call a Web Service Using SSL" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod28.asp*.
  - "How To Call a Web Service Using Client Certificates from ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod27.asp*.

Implementing Transport Layer Security with Kerberos and IPSec on Windows Server 2003

This implementation reference provides guidance on how to implement data confidentiality and data origin authentication using the Kerberos protocol and IPSec on Windows Server 2003 at the transport layer. The solution provides data confidentiality and data origin authentication between two servers hosting Web services, and another resource, such as an application server or a database.

This approach secures point-to-point communication. Messages do not require intermediaries to process them and they are not securely persisted for any period of time. Data origin authentication is done at the host layer instead of at the application or user layer. The two hosts that require data confidentiality and data origin authentication are joined to the same Kerberos realm or to different Kerberos realms that have established a cross-trust relationship.

To implement transport layer security with the Kerberos protocol and IPSec on Window Server 2003, you must perform the following tasks:

1. Implement network layer security using IPSec.
2. Configure IPSec send-and-receive policies to send and receive messages on each host to communicate with the other host that requires data confidentiality and data origin authentication.
3. Configure IPSec to use Kerberos mode authentication.

The benefits to this approach include the following:

- It provides data confidentiality and data origin authentication capabilities in one solution.
- IPSec is a well established protocol that is easy to configure and implement on the Windows Server 2003 platform.
- IPSec has very good performance compared to other solutions for data confidentiality and data origin authentication because it is below the protocol layer in the network stack.

One liability of this approach is that IPSec does not exercise very fine control over how it uses the Kerberos protocol to authenticate with another host. If business requirements exist for auditing or data origin authentication at the user or application layer, another mechanism other than IPSec must provide it.

For more information about IPSec and how to deploy it on Windows Server 2003, see "IPSec" on Microsoft.com: *http://www.microsoft.com/windowsserver2003/technologies/networking/ipsec/default.mspx*.

More Information

For information about Web Services Security, see "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)": *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.

For information about derived key tokens, see "Web Services Secure Conversation Language (WS-SecureConversation)": *http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf*.

For information about how to configure a **SqlMembershipProvider**, see "How To: Use Membership in ASP.NET 2.0" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000022.asp*.

For information about creating a custom ASP.NET 2.0 membership provider, see "Building Custom Providers for ASP.NET 2.0 Membership" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/bucupro.asp*.

For information about configuring WSE 3.0 to prevent replay attacks, see "Web Services Enhancements 3.0 <replayDetection> Element" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/b4fa188d-4804-40bd-877b-c01058555013.asp*.

For more information about performance objectives, see "Improving .NET Performance and Scalability" on MSDN: *http://msdn.microsoft.com/practices/Topics/perfscale/default.aspx?pull=/library/en-us/dnpag/html/scalenet.asp*.

For information about WSE 3.0 policy, see "Securing a Web Service" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/7b8f29da-22d5-4e03-b645-15011a80e548.asp*.

For information about Kerberos assertion policy settings, see "<kerberosSecurity> Element" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/bde6a6dd-00e4-4c37-aa8d-8821f2f25bc5.asp*.

For more information about performance objectives see, "Improving .NET Performance and Scalability" on MSDN: *http://msdn.microsoft.com/practices/Topics/perfscale/default.aspx?pull=/library/en-us/dnpag/html/scalenet.asp*.

For information about installing X.509 certificates in the local certificate store, see "How to: Use the X.509 Certificate Management Tools" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse/html/21eb7fb5-bd11-4cce-be0c-7b3d0cd14acb.asp?frame=true*.

For information about how to install X.509 certificates in the local machine certificate store, see "Certificates How To" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/ServerHelp/fb037b9f-8956-411c-a3e8-ce1dfe37da11.mspx*.

For more information about configuring the behavior of X.509 security in WSE 3.0, see "<x509> Element" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/72b7b9c9-63dd-4ce7-a25f-e40b164912d2.asp* in the WSE documentation.

For information about how to set the **findType** and **findValue** attributes for the **<x509>** element, see "<x509> Element (Policy)" in the WSE 3.0 documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/4caad727-778e-4c57-90f8-0edca69eed1f.asp*.

For information about configuring other settings for this policy assertion, see "<mutualCertificate10> Element" in the WSE 3.0 documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/973d38d8-6347-4617-983f-089e64a2b02c.asp.*

For more information about performance objectives, see "Improving .NET Performance and Scalability" on MSDN: *http://msdn.microsoft.com/practices/Topics/perfscale/default.aspx?pull=/library/en-us/dnpag/html/scalenet.asp*.

Part II

Additional Web Service Security Patterns and Guidance

In This Part:

- Resource Access Patterns
- Service Boundary Protection Patterns
- Service Deployment Patterns
- Technical Supplements

4

Resource Access Patterns

Introduction

Web services represent a programmable interface that applications often use to access resources, such as the local file system, databases, or other Web services. Distributed applications can consist of multiple interacting Web services, so you have to consider the needs and constraints of the entire application instead of focusing on a single point of interaction. Authentication, authorization, and auditing, along with other environmental and operational requirements (such as scalability requirements at each resource access interface), should combine to influence the security solution that you use to help secure access to resources.

Each of the following resource access issues involves questions you should consider:

- **Authentication credentials**:
  - Is the client authenticated and what protocol was used?
  - Does access to the resource need to be protected from direct access by authenticated clients?
- **Auditing requirements for resources**:
  - Is it sufficient to just pass the client's identity to a resource or should the client's credentials be used to access the resource?
- **Location of the resource**:
  - Is the resource located on the same computer, another computer in the same security domain, or a computer located in a different security domain?
  - Are there multiple hops involved that require the client's credentials at each interface?
- **Scalability requirements for the Web service**:
  - Does the Web service need to take advantage of resource sharing techniques such as connection pooling?

Sorting through these factors to make a decision can sometimes be a difficult task, particularly because many of them have dependencies on each other.

This chapter includes a design pattern that discusses trusted subsystem, where a trusted identity is used to access resources on behalf of a client. It also includes a technical supplement that discusses protocol transition and constrained delegation.

Important Concepts

There are some important concepts you should understand before looking at the different resource access methods. These include:

- **Credentials**. These are a set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential.
- **Identity**. This term is used throughout the discussion of resource access to represent an account associated with Active Directory.
- **Service account**. This is the Windows account that the operating system process uses when it hosts a service. Web services are usually hosted in a process managed by an application server, such as Internet Information Services (IIS) that performs operations using the identity of a service account.
- **Security context**. This is the information about an identity that allows the application of policy and rights assignment. In Windows, this translates into security roles and identifiers used for authorization.
- **Security tokens**. These are sets of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential. Most security tokens also contain additional information that is specific to the broker that issued the token.

Resource Access Methods

The following methods can be used to access resources:

- **Impersonation**. Impersonation is the act of assuming a different identity on a temporary basis so that a different security context or set of credentials can be used to access a resource. When accessing local resources, such as a file in the local file system, you need only the security context. However, when accessing resources that require authentication, such as a Web service or database, credentials are required.

- **Delegation**. Delegation is not the same as impersonation; however, it requires impersonation to work. This is a process where the service account is allowed to access a remote resource on behalf of another Windows account, which is typically the client accessing a service. Impersonation is required so that the service account can access the credentials of the account being delegated and send those credentials to the remote resource. For delegation to work, the account being delegated must also be configured to allow delegation, which is the default setting in Active Directory.

  Windows supports two types of delegation:

  - **Constrained**. This type of delegation is supported only on Windows Server 2003. This is an implementation where a service account can access only remote resources that it has been configured to access.
  - **Unconstrained**. This type of delegation is supported on Windows 2000 and Windows Server 2003. This is an implementation where a service account can use delegation to access any remote resource.

  Note: The use of unconstrained delegation is not recommended.

- **Protocol transition**. Protocol transition is a process where the service account transitions an identity that was authenticated using a non-Windows protocol into a Windows security context. This works only on Windows Server 2003 and the transitioned identity must have a valid Active Directory account. This can also be used to implement a trusted subsystem by using the service account's identity instead of the client's identity to access resources.
- **Trusted subsystem**. This is a process where a trusted business identity is used to access a resource on behalf of the client. The identity could belong to the service account or it could be the identity of an application account created specifically for access to remote resources. There are many different reasons for using a trusted subsystem, but the most common reason is to take advantage of resource sharing techniques, such as connection pooling associated with database connections.

For a detailed explanation of these resource access methods, see the "Trusted Subsystem" pattern and the "Protocol Transition and Constrained Delegation Technical Supplement" in this chapter.

There are many different protocols and techniques that can be used to authenticate with a Web service. However, from the client's standpoint, there are two distinct methods. These two methods are presenting credentials, such as the user name and password, or presenting a security token issued from a trusted source. Each of these methods has an impact on the ability to use impersonation, constrained delegation, or trusted subsystem.

There are also other security considerations that influence a decision to use impersonation, constrained delegation, or trusted subsystem, such as auditing, resource location, and scalability requirements. Impersonation is commonly used by itself; however, constrained delegation requires the use of impersonation. Typically, trusted subsystem is not used with impersonation or constrained delegation.

Table 4.1 shows how impersonation, constrained delegation, and trusted subsystem can be used based on different security considerations.

Table 4.1: Resource Access Decision Matrix

Security Consideration	Impersonation and Constrained Delegation	Trusted Subsystem
Client is authenticated using Windows authentication with Kerberos.	Both impersonation and constrained delegation can be used with a Windows account; this allows resources to be accessed using the client's identity.	Can be implemented using a trusted business identity or the service account to authenticate with the resource.
Client is authenticated using a non-Windows authentication protocol.	Impersonation and constrained delegation can be implemented using protocol transition; however, the service account must have trusted computing base (TCB) privileges.	Can be implemented using a trusted business identity or the service account to authenticate with the resource.
A service is accessing local resources and the resources are secured using ACLs based on the identity of individual clients.	Impersonation is required to access local resources using the client's identity.	Not applicable.
A service is accessing remote resources and security policy prohibits clients from directly accessing the resources.	Not applicable.	Can be implemented using a trusted business identity or the service account to authenticate with the resource.
The client's identity must be passed to resources so they can perform auditing or data entitlement.	When using impersonation or constrained delegation, the client's identity is passed using operating system capabilities to downstream resources.	The client's identity would need to be passed as part of the message header or body.
Remote resources cannot validate client credentials because the originating client does have an account in Active Directory.	Not applicable.	Can be implemented using a trusted business identity or the service account to authenticate with the resource.
The application or Web service must use resource sharing optimization techniques.	Not applicable. Most resource sharing techniques require the use of a common identity.	Using a common identity with trusted subsystem supports optimization techniques.

(continued)

Table 4.1: Resource Access Decision Matrix *(continued)*

Security Consideration	Impersonation and Constrained Delegation	Trusted Subsystem
A client is authenticated by sending the user name and password of the client in the Web service message.	The user name and password can be used to authenticate the client with Windows authentication to support both impersonation and constrained delegation.	Can be implemented using a trusted business identity or the service account to authenticate with the resource.
The client's identity must be passed to resources.	When using impersonation or constrained delegation, the client's identity is passed using operating system capabilities to downstream resources.	The client's identity would need to be passed as part of the message header or body.
Remote resources need to access another resource using the original client's credentials.	Constrained delegation must be used to flow the client's credentials to a remote resource.	Not applicable.
Resources need to perform actions based on the identity of the client.	If only the identity is required, impersonation can be used; otherwise, constrained delegation must be used.	If only the identity is required, it can be passed as part of the message header or body; otherwise, this is not an option.

The remainder of this chapter focuses on the following design pattern and technical supplement:

- Trusted Subsystem
- Protocol Transition with Constrained Delegation Technical Supplement

Trusted Subsystem

Context

A client needs to access one or more Web services that are distributed across a network. The Web services are designed so that access to additional resources (such as databases or other Web services) is encapsulated in the business logic of the Web service. These resources must be protected against unauthorized access.

Problem

How do you ensure that the client that is used to access the Web service cannot access the additional resources directly?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Security policy prohibits users from accessing downstream resources directly**. Direct access to remote resources such as a database or Web services may result in business logic being circumvented and cause data inconsistencies in underlying data stores.
- **Remote resources cannot validate user credentials**. The downstream resources may exist in a security domain that is different from the one where the client was authenticated, or the authentication protocol that was used to authenticate the client may not support delegation to the remote resource.
- **There is a risk that resources can be exploited if the Web service is compromised by an attacker**. The surface area for attackers can be reduced by restricting access to a small group of accounts. This can also simplify management of access rights for the resource.

The following condition is an additional reason to use the solution:

- **The application or Web service can take advantage of resource sharing optimization techniques**. Resource sharing optimization techniques may include connection pooling and caching.

The following condition is not resolved by the base pattern, but is resolved by Extension 1 — Flowing the Identity of the Client:

- **Resources need to perform actions based on the identity of the client**. For example, actions performed in a database may require a client identity to support data entitlement logic or to create an audit trail.

For more information, see the "Extensions" section at the end of this pattern.

Solution

The Web service acts as a trusted subsystem to access additional resources. It uses its own credentials instead of the user's credentials to access the resource. The Web service must perform appropriate authentication and authorization of all requests that enter the subsystem. Remote resources should also be able to verify that the midstream caller is a trusted subsystem and not an upstream user of the application that is trying to bypass access to the trusted subsystem.

Participants

The Trusted Subsystem pattern involves the following participants:

- **Client**. The client accesses the trusted subsystem and provides the credentials for authentication during the request to the trusted subsystem.
- **Trusted Subsystem**. A Web service that accesses the downstream resource and replaces the client's security context with its own.
- **Remote Resource**. A Web service, database or other major component of a system. Access to the remote resource is controlled to prevent unauthorized use.

Process

Figure 4.1 depicts the interactions performed when a downstream resource is accessed through a trusted subsystem.

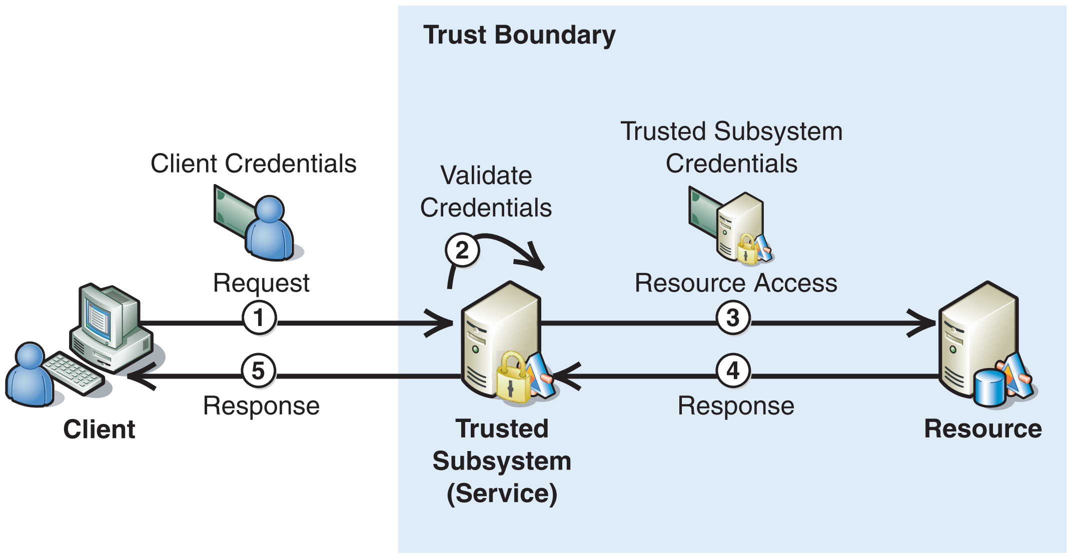


Figure 4.1
Trusted subsystem

As illustrated in Figure 4.1, the trusted subsystem process is described in the following steps:

1. The client submits a request to the trusted subsystem. The client provides credentials to the trusted subsystem.
2. The trusted subsystem authenticates and authorizes the user. Authentication can be direct or brokered. For more information, see the Direct Authentication pattern and the Brokered Authentication pattern in Chapter 1, "Authentication Patterns."
3. The trusted subsystem sends a request message to the remote resource. This request is accompanied by the credentials for the trusted subsystem (or the service account under which the trusted subsystem process is being executed).
4. The downstream resource authenticates and authorizes the trusted subsystem. It then processes the request and issues a response to the trusted subsystem.
5. The trusted subsystem processes the response and issues its own response to the client.

When multiple Web services collaborate to solve more complex problems, a Web service can simultaneously be a trusted subsystem and a resource that is accessed by a trusted subsystem. Figure 4.2 shows two overlapping trusted boundaries, with the trusted subsystem 1 taking responsibility for authenticating the client and the trusted subsystem 2 taking responsibility for authenticating trusted subsystem 1. Trusted subsystem 2's credentials are then used to access the remote resource.

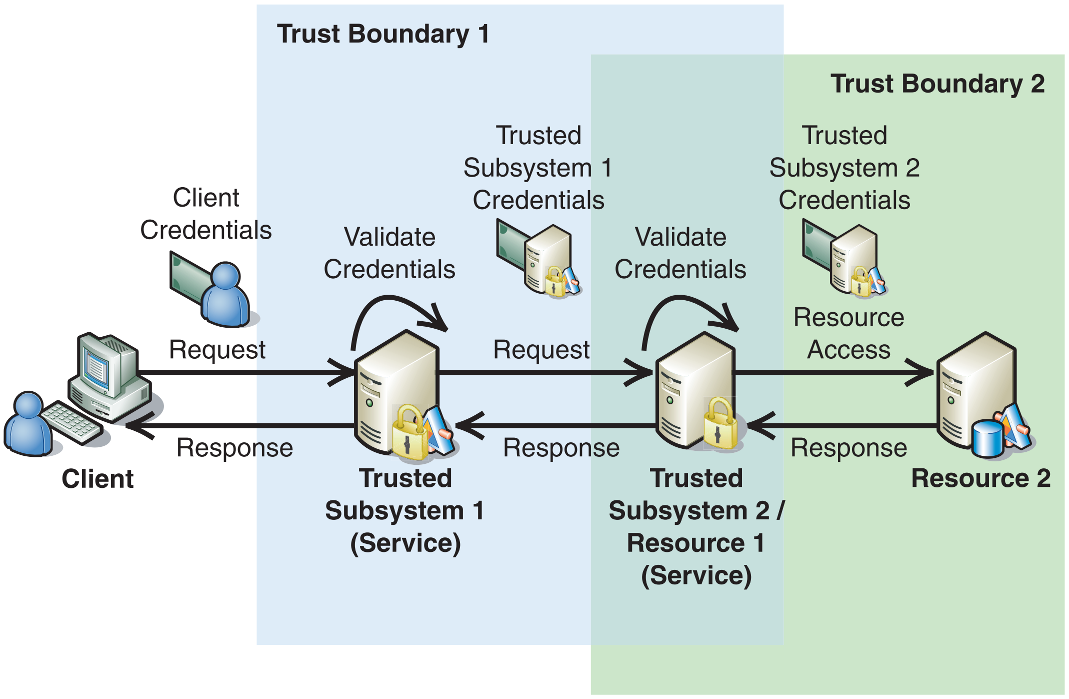


Figure 4.2
A Web service acting as a trusted subsystem and also as the resources of a trusted subsystem

Enforcing the Trust Relationship

Downstream resources must be able to verify that the midstream caller is a trusted subsystem and not just any system process. Requiring this type of verification enhances security by making it more difficult for attackers to simulate a trusted subsystem and perform man-in-the-middle attacks. Several approaches can be used to implement trusted subsystem verification:

- Authenticate the trusted subsystem with a Kerberos protocol service account.
- Use local accounts on each host.
- Use an X.509 PKI for authentication within the trusted subsystem.
- Secure communications by using IPSec between the computers in the trusted subsystem.

Kerberos Protocol Service Accounts

A common approach to implement verification with the Kerberos protocol is to use a service account that is used only within a particular trusted subsystem. This approach requires the service to be authorized so that only the trusted subsystem account can access it.

Local Accounts

When it is not possible to authenticate with a Kerberos protocol Key Distribution Center (KDC) you can create a local account on each host within the trusted subsystem. Each account has the same login and password. Accounts that are created to function this way are often referred to as mirrored accounts. While this approach provides a simple solution, it should not be your first choice. If you chose to use mirrored accounts, you should ensure that you use complex passwords and change them frequently.

X.509 PKI

An X.509 PKI can issue a certificate for each application within the trusted subsystem. The control of access to resources within the trusted subsystem is based on the ability of an application to prove possession of the certificate private key. It does this in conjunction with validating the certificate against a list of certificates that are authorized to access the resource.

IPSec

IPSec secures messages between two hosts at the network layer to provide data confidentiality, data integrity and replay detection. It can be configured to initiate secure communications with the Kerberos protocol, X.509 certificates, or a pre-shared key. IPSec performs considerably better than message layer security, but it does not allow for granular control of resources. This is because a trusted subsystem, which is established with IPSec, can only be established between the computers that participate in the trusted subsystem, and not based on a specific application accessing a specific resource.

Example

Global Bank provides a customer account client application that accesses a centralized account management database through a Web service. The client application must authenticate with the Web service to use the account management database.

In this scenario, the Web service acts as a trusted subsystem by using its own credentials to access the account management database. The client application cannot directly log in to the account management database because this violates the security policy and bypasses the business logic.

Note: This example usually requires data entitlement logic to ensure that after a customer has authenticated, he or she cannot access account details for another account. For more information, see "Extension 1 — Flowing the Identity of the Client" at the end of this pattern.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of the Trusted Subsystem pattern include the following:

- Access to the downstream resource is simplified, which allows you to take advantage of optimizations such as connection pooling to improve performance.
- Administration of access control lists on downstream resources can be simplified because only the trusted subsystem is allowed access to the resources.
- The attack surface of the Web service is reduced by limiting the resources that are authorized to access it directly.

Liabilities

If a trusted subsystem is compromised, the trusted subsystem can be used to exploit the downstream resource, potentially on behalf of a legitimate user. For this reason, trusted subsystems are often a choice target for attackers to probe for vulnerabilities. Care must be taken to ensure that a trusted subsystem is very secure.

Security Considerations

A downstream system should be able to verify that the caller is a trusted subsystem and is not an authenticated system process. This can be accomplished by establishing security claims that are issued by a trusted third party and that are verified by the downstream resource.

Extensions

The extension described here builds on the base pattern to provide additional capabilities. In addition to resolving the forces stated for the base pattern, this extension also resolves the following condition:

- **Resources need to perform actions that are based on the identity of the client**. For example, actions performed in a database may require a client identity to support data entitlement logic or to create an audit trail.

Extension 1 — Flowing the Identity of the Client

In a trusted subsystem model, the credentials of the originating client are not used for authentication purposes against downstream resources. However, in many cases, the resource must perform authorization or data entitlement checks that are based on the identity of the originating client — and not on the identity of the trusted subsystem.

You can use the following two main approaches for flowing an identity:

- The trusted subsystem provides a self-signed token.
- The trusted subsystem forwards a signed token that is provided by an authentication broker.

Note: As with any data that is passed from a trusted subsystem to a downstream resource, the downstream resource relies on the integrity of the trusted subsystem. In each approach described here, the client's identity is simply flowed as part of the message from the trusted subsystem. It is not possible to detect if the trusted subsystem substituted one user's signed or unsigned credentials in place of alternative (perhaps cached) credentials for malicious reasons.

Approach 1 — The Trusted Subsystem Provides a Self-Signed Token

With this approach, the identity that the trusted subsystem sends to the resource is included in the message. You can include the identity in the message the following two ways:

- Include the client's identity as metadata in the SOAP message header. This can be performed by using a custom SOAP header or by using a WS-Security **UsernameToken** without a password.
- Include the client's identity in the body of the message.

In both cases, the message (including the client's identity) must be signed by the trusted subsystem, so that the downstream resource can authenticate the trusted subsystem and perform data origin authentication. The resource must assume that the trusted subsystem has authenticated the client whose identity is contained in the message. Otherwise, it has no way to know directly that the client has been authenticated.

Approach 2 — The Trusted Subsystem Forwards a Signed Token That Is Provided by a Trusted Third Party

With this approach, the trusted subsystem is responsible for forwarding a token to the downstream resource that is signed by a trusted third party, such as a Security Token Service (STS). The downstream resource can then validate the client's claims within the token, based on the trust relationship with the STS. It also allows the resource to verify that the client was authenticated recently. For this reason, the tokens issued by the STS should have a short lifetime.

When the client authenticates with the STS, the STS issues a signed security token that contains claims, such as the client's identity and roles. The token is used by the client to authenticate with the trusted subsystem. After the trusted subsystem receives the security token and authenticates the client, it signs the token and forwards it in a signed message to the downstream resource. The downstream resource authenticates the trusted subsystem and it is also able to verify the clients token using the signature of the STS within the forwarded token.

Protocol Transition with Constrained Delegation Technical Supplement

Consider the following scenario:

> *You are deploying a Web service that does not use Windows integrated authentication. After a client is authenticated; the client needs to be transitioned to a Windows account so that role-based authorization can be performed. The Web service also needs to interact with Web services or other downstream resources that can only be accessed with valid Windows credentials.*

The common approach to this problem is to have the client application send a user ID and password that can be used for authentication within the Web service. However, this requires the client application to store a password for use when it accesses the Web service. In addition, the password needs to be protected while it is in transit between the application and the Web service. Both of these requirements represent a security risk that should be avoided.

The solution to this problem is to use the new Kerberos protocol extensions in Windows Server 2003. The new extensions require the user ID but not the password. You still need to establish trust between the client application and the Web service; however, the application is not required to store or send passwords. One of these extensions, referred to as Protocol Transition, can initialize a valid **WindowsIdentity** object with only the user ID. The other extension uses the new **WindowsIdentity** object with constrained delegation to access remote resources.

The new extensions are:

- The Kerberos protocol transition extension, S4U2Self.
- The Kerberos constrained delegation extension, S4U2Proxy.

The following list identifies three distinct operations that you can implement with the new extensions:

- Use protocol transition to initialize a **WindowsIdentity** object for authorization checks.
- Use protocol transition to initialize a **WindowsIdentity** object for impersonation.
- Use constrained delegation to access remote resources.

The first two operations can be implemented independently of each other, but the third operation requires that you use protocol transition when you are not using Kerberos authentication. In other words, constrained delegation has two configurations; one that requires Kerberos authentication and another that works with any authentication protocol. When you use the configuration that supports any authentication protocol, you must first implement protocol transition with impersonation before you implement constrained delegation.

The next section provides details on the extensions themselves. After you have an understanding of these extensions, see the "Implementation" section to learn how to implement the operations described earlier.

New Kerberos Extensions

As previously mentioned, Windows Server 2003 provides two new Service-for-User (S4U) Kerberos extensions that support protocol transition and constrained delegation. Protocol transition and constrained delegation can be used independently of each other, but they are often used together to implement the scenario described in the introduction.

Protocol Transition

The S4U2Self Kerberos extension can be used to initialize a **WindowsIdentity** object with the user ID with a valid Windows account in Active Directory. The password associated with the user ID is not required. This feature allows you to transition from any authentication protocol into the Kerberos authentication protocol.

This operation is accomplished by using the ticket-granting ticket (TGT) of a service account to request a service ticket for itself. The service account in this case is the one associated with the Web service that performs the protocol transition. The service ticket that is returned from the ticket-granting service (TGS) contains identity and principal information for the user whose ID was sent with the request.

The new **WindowsIdentity** that is initialized with this service ticket can then be used to perform role-based authorization checks. In addition, when used with constrained delegation, this new identity can be used to access downstream resources. There are limitations to what this new identity is allowed to do that are based on the privileges of the service account. These limitations are discussed in the "Implementation" section later in this technical supplement.

Constrained Delegation

The S4U2Proxy Kerberos extension provides an implementation of constrained delegation that allows you to use a Kerberos service ticket — instead of a TGT — to request another service ticket. Delegation is considered to be constrained because the identity (service account) that is used to request the service ticket must be configured to access a specific service.

Note: For more information about the use of TGTs, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

Constrained delegation works with or without protocol transition. The primary restriction is that the service account used to request a Kerberos service ticket must be configured to access the requested service. In addition, the service account must be able to impersonate the client prior to calling the service. For example, when you use Windows integrated authentication with impersonation, the default Web server's computer account can be configured for constrained delegation without making any changes to the Internet Information Services (IIS) process account.

A restriction of protocol transition is that the Web server's computer account cannot be used for constrained delegation without modifying the IIS process account. The reason for this is that the default IIS process account (which is the NT AUTHORITY\NETWORK SERVICE account on Windows 2003 Server) does not have necessary privileges to implement impersonation using the **WindowsIdentity** object that was created during protocol transition. Instead of modifying the default IIS process account, you can also use a different service account for the IIS process.

Note: The S4U2Self and S4U2Proxy extensions are only supported on Windows Server 2003. As a result, protocol transition with constrained delegation does not work on Windows Server 2000 or on Windows XP. It is possible, however, to call services on these platforms by using the service ticket that is retrieved from the delegated request.

For more information about the Kerberos protocol and related patterns, see the following:

- Brokered Authentication: Kerberos in Chapter 1, "Authentication Patterns"
- Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security"
- Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements"

Scenarios

.NET Framework applications can implement protocol transition by creating an instance of the **WindowsIdentity** object with a User Principal Name (UPN), which is similar to an e-mail address. For example, if the user ID is **steve** and the corresponding Active Directory domain is **globalbank.net,** the UPN is **steve@globalbank.net**.

It is also possible to use protocol transition to initialize a **WindowsIdentity** object using a common Active Directory account for trusted subsystem implementations. This type of approach is normally used when you want to improve scalability with resources that use object or connection pooling based on the credentials that were used to access them. For example, connection pooling with SQL Server will work only if a common identity is used.

As a result, the following two primary scenarios are associated with protocol transition in Windows:

- Transitioning from a different authentication protocol, such as X.509 client certificates, into the Kerberos protocol.
- Transitioning from custom authentication by using a common identity for trusted subsystem implementations.

Implementation

This section describes how to implement each of the following three distinct operations that you were introduced to earlier in this technical supplement:

- Use protocol transition to initialize a **WindowsIdentity** object for authorization checks.
- Use protocol transition to initialize a **WindowsIdentity** object for impersonation.
- Use constrained delegation to access remote resources.

These operations are performed with a sample application that starts with authorization and finishes with the use of impersonation and constrained delegation to access a remote resource.

Instead of focusing on client authentication, the discussion focuses on protocol transition with constrained delegation by using an identity that is retrieved from an X.509 client certificate. For more information about using client certificates for authentication with Web services, see How to Call a Web Service Using Client Certificates from ASP.NET.

Note: This guidance assumes that the reader is familiar with Active Directory, Internet Information Services (IIS), and the .NET Framework.

Use Protocol Transition to Initialize a WindowsIdentity Object for Authorization Checks

Starting with the .NET Framework 1.1, a new constructor was added to **WindowsIdentity** that uses the S4U2Self Kerberos extension to request a service ticket. The ticket-granting ticket (TGT) of the service account is used to request a service ticket for itself by using identity information from the client who is accessing the service. As a result, the privileges of the service account also affect the type of **WindowsIdentity** object that is created. For example, if the service account has Trusted Computing Base (TCB) privileges, the **WindowsIdentity** object can be used for impersonation, which is required to implement constrained delegation.

Even if the service account does not have TCB privileges (which is often the case), you can use the new **WindowsIdentity** constructor to initialize an identity object and then use that to initialize a **WindowsPrincipal** object for role-based authorization checks, using the client's security roles. A service account without TCB privileges can also be used to access resources directly. However, that account's identity is used to access the resource instead of the client's identity.

The following code example shows how to initialize a **WindowsIdentity** object with the user ID and domain information that is associated with the client. It then uses that information to perform role-based authorization checks.

```
WindowsIdentity identity = new WindowsIdentity( <logon name>@<domain> );
if( identity != null )
{
    WindowsPrincipal userPrincipal = new WindowsPrincipal( identity );
    if( userPrincipal.IsInRole( @"GLOBALBANK\ServiceUsers" ))
    {
        ...
    }
    else
    {
        lblMessage.Text = "Not In Role: Service access denied";
    }
}
```

Example: Using protocol transition to initialize a WindowsIdentity object for role-based authorization checks

It is useful to have access to a client's identity and roles, but the default IIS service account does not have the necessary permissions to impersonate the client when it accesses a resource. To provide this functionality, the next section describes how to configure a new service account that will be used as the identity of an application pool in IIS 6.0.

Note: The previous code example from a Web application shows what was used to implement protocol transition with constrained delegation. **GLOBALBANK\ServiceUsers** represents an Active Directory group that is used to provide role-based authorization checks. The field named **lblMessage** is a Web application **Label** control that is used to display messages. This example is extended throughout the remainder of this technical supplement.

Use Protocol Transition to Initialize a WindowsIdentity Object for Impersonation

If you want to use the client's security context to access resources, impersonation must be implemented prior to accessing the resource. To implement this with protocol transition, you should create a new service account and configure that account to perform the protocol transition.

By default, IIS applications and services run under the NETWORK SERVICE account on Windows Server 2003. The easiest way to support protocol transition is to give this account Trusted Computing Base (TCB) privileges on the service host. An account with TCB privileges can act as part of the operating system when it performs operations. However, the problem with this approach is that the NETWORK SERVICE account is used by many Web applications and services. Giving this account operating system privileges represents a significant security risk.

With the use of application pools in IIS 6.0, you can mitigate this risk by creating a new pool that uses an identity with TCB privileges. To accomplish this task, you must first create a domain user account and configure it to have proper privileges on the Web server. After the account is configured, it can be used as the identity of a new application pool. Any Web applications or services that need to implement protocol transition with impersonation can then use this new application pool.

Note: The following steps require that you have administrative privileges on appropriate servers to perform the operations.

Step One: Create a Domain Account

On the domain controller, use the following steps to create a new user account.

► **To create a domain user account**

1. On the Administrators menu, click Active Directory Users and Computers.
2. Create a new user and configure it with the following settings:

   First Name: Domain
   Last Name: Pool
   User Name: DPool
   Clear: User must change password at next login
   Select: User cannot change password
   Select: Password never expires

The new account is automatically added to the Users group on the domain. You do not need to add it to any other groups. At this point, there is nothing else you need to configure for the account on the domain, but this account does need additional privileges on the host Web server.

Step Two: Configure the Domain Account on the Web Server

Several permissions are required on the Web server to use the new domain account for protocol transition with IIS 6.0. You must configure the account for TCB privileges and add it to a group that has permissions for application pools. To work correctly, the account also needs special permissions on a temporary folder for protocol transition.

► **Assign TCB privileges**

1. On the Administrators menu, click Local Security Policy.
2. Expand **Local Policies**, and then click **User Rights Assignments**.
3. Open the **Act as part of the operating system** policy, and add the DPool account that you created in the previous step.

► **Add account to IIS_WPG**

1. On the Administrators menu, click Computer Management.
2. Expand **Local Users and Groups**, and then click **Groups**.
3. Open the **IIS_WPG** group, and then add the DPool account.

► **Give IIS_WPG special folder permissions**

1. Open Windows Explorer, and then click the **%SYSTEM%\Temp** folder.
2. Right-click the Temp folder, and then click **Sharing and Security**.
3. On the **Security** tab, click the **Advanced** button.
4. In the **Advanced Security Settings for Temp** dialog box, click the **Add** button and add the **IIS_WPG group**. This opens the **Permission Entry for Temp** dialog box.
5. In the **Permission Entry for Temp** dialog box, select the following check boxes:

   List Folder / Read Data
   Delete

This last configuration is required to support protocol transition, but it makes sense to assign these rights to the IIS_WPG group instead of the individual domain accounts. In addition, because the NETWORK SERVICE account already has this privilege, you are not assigning any privileges to IIS_WPG that a typical Web application does not have.

Step Three: Create a New Application Pool

This step uses the Internet Information Services (IIS) Manager, which is located on the **Administrative Tools** menu.

► **To add a new application pool**

1. Expand the server (local computer), and then click **Application Pools**.
2. Right-click **Application Pools**, point to **New**, and then click **Application Pool**.
3. Name the pool **DomainPool**. Make sure the **Use default settings for new application pool** option is selected, and then click **OK** to create the new pool.
4. In IIS Manager, expand **Application Pools**, and then click the new pool that you created.
5. Right-click **DomainPool**, and then click **Properties**.
6. On the **Identity** tab, click **Configurable**, and then type the new DPool domain account you created in Step One: Create a Domain Account.
7. Click **OK** to close the dialog box.

Step Four: Configure the Web Application to Use the New Application Pool

This step assumes that you have an ASP.NET Web application or service that uses protocol transition to access a Web service that requires message-based Kerberos authentication. The creation of the Web applications and implementation of message-based security with the Kerberos protocol are beyond the scope of this chapter. Instead, the focus is on tasks that are required to implement protocol transition.

► To configure the Web application to use the new application pool

1. In the **Internet Information Services (IIS) Manager** dialog box, expand the server (local computer), expand **Web Applications**, expand the Web site folder, and then click your Web application.
2. Right-click the Web application, and then click **Properties**.
3. On the **Virtual Directory** tab, click the application pool (**DomainPool**) that was created in the previous step.
4. Click **OK** to close the dialog box.

With the Web application configured to use the new application pool, you can use the following code example to test the configuration and make sure that impersonation is supported by the **WindowsIdentity** object created using protocol transition.

```
...
    WindowsIdentity identity = new WindowsIdentity( <logon name>@<domain> );
    if( identity != null )
    {
        WindowsImpersonationContext context = null;
        try
        {
            context = identity.Impersonate();
            // Perform operations that require impersonation...
        }
        catch( Exception ex )
        {
            lblMessage.Text = "Impersonation Failed: " + ex.Message;
        }
        finally
        {
            context.Undo();
        }
    }
...
```

Example: Using protocol transition to initialize a WindowsIdentity object for impersonation

Notice that the **Impersonate** operation is performed within a **try/catch** block. This is because the **WindowsIdentity** object does not provide information about the service ticket that is associated with the service account. In other words, the identity cannot be checked to determine whether it supports impersonation before attempting the **Impersonate** operation. This means that if the service account does not have TCB privileges, this operation will throw an exception.

Note: The code sample performs a **context.Undo()** statement in the **finally** block to revert the security context back to the original identity. If this operation fails, which is rare, the recommendation is to exit the application with a system error and shut down the process. In other words, the application process should be stopped immediately.

Even though you now have IIS configured for protocol transition with impersonation, you still cannot access downstream resources with the transitioned identity. Your final task is to configure Active Directory to support constrained delegation.

Use Constrained Delegation to Access Remote Resources

Two types of constrained delegation are available: one that requires the Kerberos protocol and another that supports any protocol. To support protocol transition, you must use the configuration that supports any authentication protocol. However, before configuring delegation, you need to create a Service Principal Name (SPN) for the DPool domain account, which is the account you created in the previous task.

Step One: Create SPN for Domain Account

A Service Principal Name (SPN) represents a unique name that is used by the Kerberos protocol to access a service's long term key when it creates a service ticket. To request a service ticket, the service must have an associated SPN registered in Active Directory. By default, all of the standard services use a HOST-based SPN, which is configured when the operating system is installed. By using a different account as the identity for the application pool, this host SPN can not be used. Instead you need to create a new host-based SPN for the domain account.

The tool you use to create a new SPN is named setspn.exe, which is found in Windows Support Tools for Windows Server 2003. To create a new SPN, open the command prompt in the **Support Tools** menu and type the following command.

```
setspn -a http/<host>.<domain> DPool
```

The following list explains the command elements:

- **-a** tells SetSPN to create and then add a new SPN.
- **http** is the built-in service class used by IIS and Internet Explorer when Windows integrated security is used.
- **<host>** is the name of the host computer; you need to change this to match the Web server you are configuring.
- **<domain>** is the domain; you need to change this to match your domain.
- DPool is the host user account configured for protocol transition.

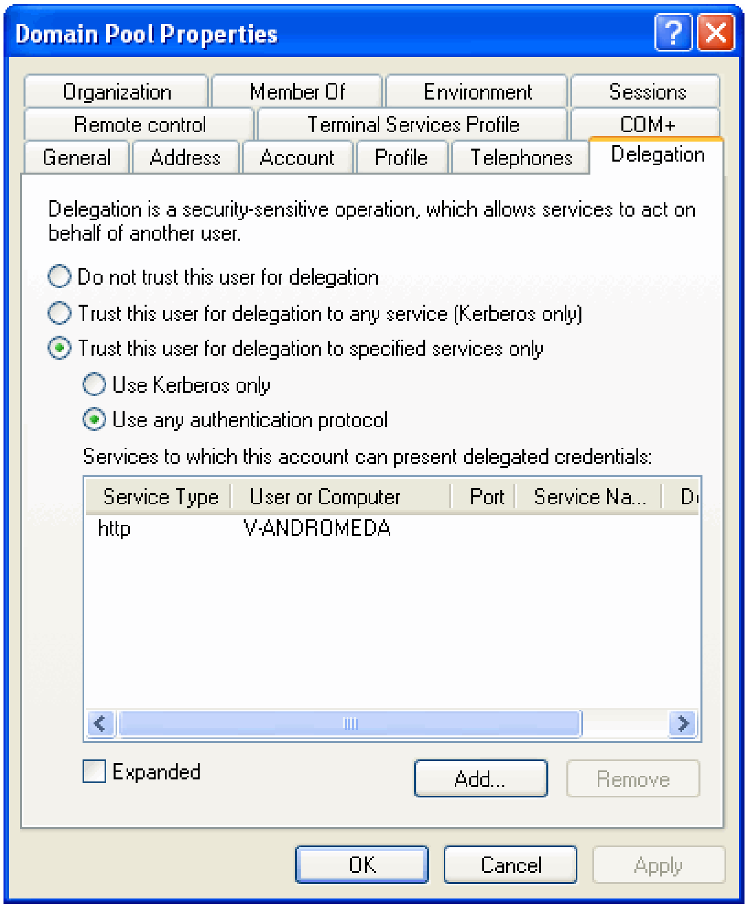
For more information about SPNs, setspn.exe, and the Kerberos security protocol, see Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

After the SPN is created, it is possible to configure constrained delegation so that the Web application or service can use a transitioned identity to access downstream resources.

Note: The example in this chapter uses the built-in HTTP service class as the SPN that maps to a domain account. This is not necessary in message layer security where you can specify the SPN. However, when you use Windows Integrated Security, you must map the service account to the HTTP SPN because both Internet Explorer and IIS use the HTTP SPN when they interact with a Web-based service.

Step Two: Configure Delegation

When the SPN is created, a new **Delegation** tab is added to the associated domain account, which provides the ability to configure constrained delegation. Figure 4.3 shows the new **Delegation** tab for the DPool account.



Delegation is a security-sensitive operation, which allows services to act on behalf of another user.

- ◯ Do not trust this user for delegation
- ◯ Trust this user for delegation to any service (Kerberos only)
- ◉ Trust this user for delegation to specified services only
  - ◯ Use Kerberos only
  - ◉ Use any authentication protocol

Services to which this account can present delegated credentials:

Service Type	User or Computer	Port	Service Na...	D
http	V-ANDROMEDA			

☐ Expanded

Figure 4.3
The ***Domain Pool Properties*** *dialog box*

As you can see in Figure 4.3, the Domain Pool (DPool) properties are configured to **Trust this user for delegation to specified services only** and for **Use with any authentication protocol**. In the **Services to which this account can present delegated credentials** list, there is a single entry with a service type of HTTP, which means that DPool can only access the HTTP service class that is associated with the user or computer shown in the next column.

Remember that DPool is configured as the identity of an application pool in IIS. This application pool is used to host Web applications that implement protocol transition. In other words, this is the identity that is used to request a Kerberos service ticket on behalf of a user that was transitioned into the Kerberos protocol. To request a ticket on behalf of the transitioned user, you must configure DPool to support constrained delegation with any authentication protocol as shown in Figure 4.3.

Note: You can use constrained delegation to access any service that supports the Kerberos protocol. This means that a Web application that is running on Windows Server 2003 can access a Web service that is running on Windows XP by using protocol transition with constrained delegation.

Sample Code

After you complete all of the tasks, you can implement protocol transition from any Web application or service that is hosted in the new application pool. However, the main restriction is that you can only access services that are configured in the **Delegation** tab of the host identity that is associated with the application pool. This identity is a domain account that has been configured to support protocol transition and constrained delegation on the Web server.

The following code sample demonstrates how to implement protocol transition with constrained delegation using a client's logon name.

```
WindowsIdentity identity = new WindowsIdentity( <logon name>@<domain> );
if( identity != null )
{
    WindowsPrincipal userPrincipal = new WindowsPrincipal( identity );
    if( userPrincipal.IsInRole( @"GLOBALBANK\ServiceUsers" ))
    {
        WindowsImpersonationContext context = null;
        try
        {
            context = identity.Impersonate();
            try
            {
                Service.KerberosService service =
                    new Service.KerberosService();
                service.PreAuthenticate = true;
                service.Credentials = CredentialCache.DefaultCredentials;
                lblMessage.Text = service.HelloWorld();
            }
```

(continued)

(continued)

```
            catch( Exception ex )
            {
              lblMessage.Text = "Service Call Failed: " + ex.Message;
            }
        }
        catch( Exception ex )
        {
            lblMessage.Text = "Impersonation Failed: " + ex.Message;
        }
        finally
        {
            context.Undo();
        }
    }
    else
    {
        lblMessage.Text = "Invalid Role: Service access denied";
    }
}
```

Example: Using protocol transition with constrained delegation to access a service with the client's logon name.

The Web service in this example was deployed on Windows XP and has been configured to support Windows integrated authentication with impersonation enabled. When a Web service request is sent with Windows integrated authentication, the two bold lines in the previous code example must be used to attach the credentials to the message. Setting **PreAuthenticate** to **true** adds the credentials to the initial request, which prevents a round trip. The **CredentialCache** holds the credentials of the identity that was impersonated.

When you use message layer security, the lines in bold are not required. For an example of message layer security using the Kerberos protocol, see Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Implementation Context

This section describes some of the more significant benefits, liabilities, and security considerations of implementing protocol transition with constrained delegation.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered.

Benefits

The benefits of protocol transition with constrained delegation include the following:

- Access to the downstream resource is based on the identity of the client, and can therefore be traced back to the original client of the online application. This allows granular auditing and authorization that are based on the originating client's identity instead of the identity of the online application or service.
- An online application that does not support protocols that are normally used on the internal network can transition clients into a protocol that is supported.
- With protocol transition, it is possible to implement delegation and impersonation without storing a client's password on the Web server.

Liabilities

The liabilities of protocol transition with constrained delegation include the following:

- When impersonation or constrained delegation is used, it may not be possible to take advantage of optimizations, such as connection pooling. Most resource-sharing optimizations require the use of a common identity when they authenticate with the resource. For example, a separate connection exists for each client when you use impersonation or constrained delegation to access a database. This prevents the ability to share connections with multiple clients (connection pooling).
- Configuring a domain account to use the host-based HTTP SPN means that other Web applications and services on that Web server must also use the same application pool to support any authentication request that uses the HTTP SPN, which is the default behavior with Windows integrated authentication.
- If the account that is used to implement protocol transition is compromised, it is possible for an attacker to use any Active Directory account when he or she accesses resources. However, the number of resources is restricted because an identity that is created with protocol transition must use constrained delegation to access a resource.

Security Considerations

Security considerations associated with protocol transition with constrained delegation include the following:

- Establishing a security context for a client without proving its identity requires a high degree of trust in the system that performs the protocol transition. Because of its trusted responsibilities, the system that performs protocol transition is likely to be a high-interest target for attackers. You should mitigate this threat by limiting access to private networks and using a separate application pool with an identity that is configured for protocol transition.

- You cannot use constrained delegation across a domain boundary. Constrained delegation is restricted to services in a single domain. All domain controllers in the domain must run Windows Server 2003, and the domain must operate at the Windows Server 2003 functional level.

More Information

For more information about Web services security, see "Web Services Security" in *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch10.asp*.

For more information about using impersonation and delegation in ASP.NET 2.0, see "How To: Use Impersonation and Delegation in ASP.NET 2.0" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000023.asp*.

For more information about designing the authentication and authorization mechanisms for a distributed ASP.NET Web application, see "Authentication and Authorization" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod03.asp*.

For more information about developing identity-aware applications, see "Developing Identity-Aware ASP.NET Applications, Identity and Access Management Services" on MSDN: *http://www.microsoft.com/technet/security/topics/identitymanagement/idmanage/P3ASPD_1.mspx*.

For more information about the Kerberos protocol extensions, see "Exploring S4U Kerberos Extensions in Windows Server 2003" on MSDN: *http://msdn.microsoft.com/msdnmag/issues/03/04/SecurityBriefs/default.aspx*.

For more information about implementing protocol transition and constrained delegation, see "Kerberos Protocol Transition and Constrained Delegation" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/constdel.mspx*.

For more information about using the Kerberos protocol extensions, see "How To: Use Protocol Transition and Constrained Delegation in ASP.NET 2.0" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/paght000024.asp*.

For more information about using client certificates for authentication with Web services, see "How to Call a Web Service Using Client Certificates from ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetHT13.asp*.

5

Service Boundary Protection Patterns

Introduction

Chapter 2, “Message Protection Patterns,” described how to provide protection against data tampering and unauthorized access to message content. However, in many cases you will need to provide additional protection at the service’s boundary to:

- Protect Web services against malformed or malicious content.
- Ensure that when a Web service operation fails you do not accidentally reveal confidential information in the SOAP Fault that is returned.
- Prevent an attacker from intercepting a message and replaying it to force a Web service operation to execute multiple times.

This chapter describes how to provide service boundary protection. It includes the following design and implementation patterns:

- Message Replay Detection
- Implementing Message Replay Detection in WSE 3.0
- Message Validator
- Implementing Message Validation in WSE 3.0
- Exception Shielding
- Implementing Exception Shielding

Message Replay Detection

Context

A client calls a Web service by sending messages across a public network. When the Web service processes the messages, data is updated or business processes are initiated. If a message that is intended for one of these Web services is intercepted and replayed, there is a risk that the same operation might be performed multiple times.

Problem

How do you protect a service from an attacker who replays an intercepted message?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **A replayed message will cause data inconsistency**. This can have a negative impact on business operations and cause financial damage or legal liability. For example, if funds are transferred between bank accounts multiple times, the balance of each party's account will be altered.
- **Messages traverse intermediaries on the network, where the intermediaries are not trusted**. When messages traverse untrusted intermediaries, they can be intercepted and replayed after the initial relay of the message. This form of attack is possible even if message protection techniques, such as data origin authentication and data encryption, are used to protect against tampering of data and unauthorized access to data.

The following condition is an additional reason to use the solution:

- **The Web service is susceptible to message flooding denial of service attacks from message replay**. If the normal functions of a Web service are system-intensive or network-intensive, an attacker can cause a bottleneck in the service by launching an automated attack that rapidly replays intercepted messages in large quantities. This reduces the availability of the service.

Solution

Cache an identifier for incoming messages, and use message replay detection to identify and reject messages that match an entry in the replay detection cache.

Message replay detection requires that individual messages can be uniquely identified. This ensures that a legitimate message is not rejected because of a match in the replay detection cache. Message replay detection also requires that messages have not been tampered with in transit. This ensures that the replay detection cache does not accept messages that have been captured and modified by an attacker.

Messages signed using a WS-Security XML signature must include a **SignatureValue** element, which can be cached as an identifier for the message. The **SignatureValue** is computed from hash values of the message parts that are being signed, including the message body and the timestamp.

Note: A **SignatureValue** is not truly unique because it runs the theoretical risk of collision (where the same value can be unintentionally reproduced). In most cases, the risk is very low, so **SignatureValue** is an appropriate choice for a message identifier.

The **SignatureValue** element is added to the cache, along with a timestamp from the server, indicating the time it processed the message. This allows entries to be cleared from the cache at regular intervals and to not accumulate indefinitely. The service can be designed to automatically reject incoming messages that arrive on or after a defined acceptable time delay.

Participants

The Message Replay Detection pattern involves the following participants:

- **Client**. The client accesses the Web service.
- **Service**. The service is the Web service processes requests received from clients. The service implements the replay detection logic.
- **Replay cache**. The replay cache is the entity that caches the incoming messages with a unique identifier to detect the replay messages.

Process

Figure 5.1 illustrates the process of sending a message to a Web service that implements replay detection.

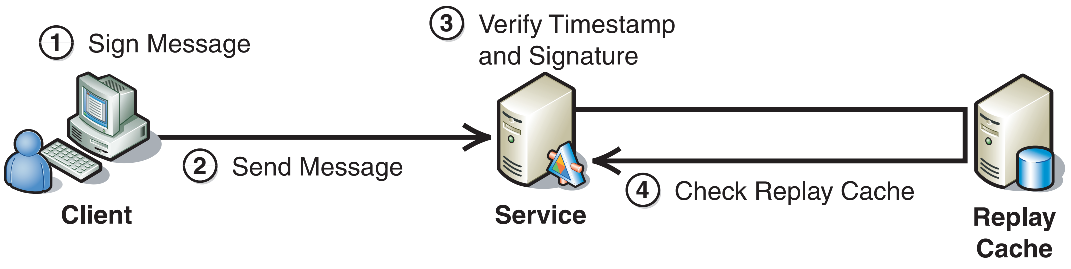


Figure 5.1
A Web service implementing message replay detection

As illustrated in Figure 5.1, the process of a Web service implementing message replay detection is described in the following steps:

1. **The client signs the message**. This signature provides assurance that the message has not been altered in transit. For more information about data integrity, see Data Origin Authentication in Chapter 2, "Message Protection Patterns."
2. **The client sends the signed message to the recipient**.
3. **The service verifies the client's signature and the message timestamp**. The Web service verifies the message signature to ensure that the message contents have not been altered in transit. If the message signature is valid, the Web service compares the message timestamp to its own current clock value. If either the signature is invalid or the message was received beyond the acceptable time span, the message is rejected.
4. **The service checks the replay cache for the SignatureValue field**. The Web service checks the replay cache for the **SignatureValue** that is used to uniquely identify the incoming message. If the **SignatureValue** is already in the cache, the message is rejected as a duplicate. If the message signature is not in the cache, the message signature and timestamp are added to the cache.

Note: The Web service must be designed to accept messages that are no older than the messages that have already been removed from the cache. Otherwise, an attacker will be able to replay a message that was previously cleared from the replay cache.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

Messages cannot be replayed, either accidentally or for malicious intent. Any attempt to replay an intercepted message will result in the message being rejected by the service. Any attempt by the attacker to tamper with the message to spoof the replay mechanism will invalidate the message signature, which causes the service to reject it.

Liabilities

The liabilities associated with the Message Replay Detection pattern include the following:

- The Web service must carefully manage its replay cache to balance scalability and security by clearing the cache at regular intervals.
- If the service is deployed to more than one server in a Web farm, a common replay cache must be used for the cache to be effective. A database is often used for this purpose, but using a database can increase latency of processing messages. The database itself might also be susceptible to denial of service attacks, if an attacker floods connections to the database that is maintained in the Web farm. To help mitigate this issue, you should consider deploying the database server with failover support and ensure that connections to the database server are carefully managed within the Web farm.

Security Considerations

Security considerations associated with the Message Replay Detection pattern include the following:

- The unique identifier for the message must be saved in the replay cache prior to processing the request. This prevents concurrency issues if a second message arrives before the first message has finished executing.
- Some of the steps performed while attempting to detect replayed messages can adversely affect system response time. For example, verifying the signature on the identifier and timestamp is computationally intensive. Reading or updating the replay cache can also impact response time if the cache is on a different computer than the recipient.
- Preventing message replay can help stop a denial of service attack from accessing resources, but it is also possible for an attacker to launch a denial of service attack on the computer that is using message replay detection. The attacker does this by replaying a large number of messages to exploit high resource consumption. To minimize the impact of the attack on system availability and response time, it is important to ensure that the service implements replay detection as efficiently as possible.
- A clock skew value (TTL in seconds) is set on the server to determine the acceptable clock skew between the client and the service. If a message is received outside the acceptable time range, the message will be rejected, even if it is not already present in the cache. Therefore, it is important to ensure that clocks are closely synchronized between the sender and the recipient. This is best achieved by using time synchronization services, with the sender and recipient synchronizing their local clocks to a centralized source. The clock skew must always be less than the time that the messages are held in the cache; if it is not, a replayed message may be accepted because it will already have been deleted from the cache.

- In some cases, a client may not receive a response from a service. As a result, the client will not know whether the request succeeded. A common example that afflicts e-commerce transactions is where a user clicks the **Submit** button twice on a Web form. This scenario requires a different approach, such as the one described in the Idempotent Receiver design pattern. For more information about idempotent Web services, see Idempotent Receiver on the Enterprise Integration Patterns Web site.
- The length of time that messages should be held in the cache varies, depending on the specifics of the recipient. If an application receives a very large number of messages per second, the cache lifetime may be very small, perhaps only a few minutes. In other cases, the cache lifetime may be significantly longer, perhaps hours or days.

Variants

XML signatures provide a basis for effective message identifiers that support message replay detection. They are particularly useful where end-to-end security is required. However, there are alternative ways to implement message replay detection. You can use the following alternatives to XML signatures:

- **Use the full message**. The message itself is unique because the message header contains a timestamp and an XML signature. However, caching the full message can be inefficient because the cache might need to be very large.
- **Use an identifier that is unique to a session, such as a sequence ID**. In this case, each message is assigned a sequence number that is unique within the scope of the active session. This approach requires the client and the server to be synchronized and requires the server to maintain some form of session state to communicate with the client. The session scope may be defined by a span of time that is agreed on by both parties, after which the session must be renewed or renegotiated. Session scope can also be defined by the validity period of a security token used to establish secure communications between the two parties. Both Kerberos and SSL use session-based sequence numbers in their respective replay detection mechanisms.

Related Patterns

Two types of patterns are related to this pattern: a child pattern and a pattern that the Message Replay Detection pattern uses.

The following child pattern is related to the Message Replay Detection pattern:

- **Implementing Message Replay Detection in WSE 3.0**. It provides steps and recommendations to implement message replay detection at the message layer by using WSE 3.0.

The Message Replay Detection pattern uses the following pattern:

- **Data Origin Authentication**. The Data Origin Authentication pattern demonstrates how messages are signed to verify that they are from the intended recipient and have not been altered in transit.

Implementing Message Replay Detection in WSE 3.0

Context

You are implementing a Web service that uses Web Service Enhancements (WSE) 3.0. The Web service accepts messages sent across a public network from clients that manipulate sensitive data or initiate business processes. You need to ensure that the Web service does not process a message that has been intercepted and replayed by an attacker in an attempt to access or manipulate the sensitive data.

Objectives

The objectives of this pattern are to:

- Prevent the service from accepting and processing messages that have expired, while allowing for clock skew.
- Prevent the service from accepting and processing messages that attackers have replayed.
- Support replay attack detection for Web services deployed in a Web farm through a database-supported replay cache.
- Demonstrate an implementation of message replay detection using a WSE 3.0 custom assertion.

Content

This pattern consists of the following sections:

- **Implementation Strategy**. This section provides a high-level description of the strategy used to implement the Message Replay Detection pattern.
- **Implementation Approach**. This section describes the steps required to implement this pattern:
  - Configure the client
  - Configure the service
- **Resulting Context**. This section outlines the benefits, liabilities, and security considerations related to the pattern.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

This document provides steps and recommendations to implement message replay detection at the message layer using WSE 3.0.

Use a custom policy assertion to verify that the service has not previously accepted and processed an incoming message by maintaining a message replay cache. The custom policy assertion implements the following logic:

- Incoming messages are recognized by a message identifier that the policy assertion implements. The message identifier is contained in the **<SignatureValue>** element of the message signature.
- If the message identifier for an incoming message is not in the cache, the service has not processed the message within the lifetime of the cache, and the identifier is added to the cache.
- If the message identifier is in the cache, the message is rejected as a replayed message.

Note: To fully understand this pattern, you must have some familiarity and experience with the .NET Framework, WSE 3.0 policy assertions, and Web service development.

Participants

The Message Replay Detection pattern involves the following participants:

- **Client**. The client accesses the Web service.
- **Service**. The service is the Web service processes requests received from clients. The service implements the replay detection logic.
- **Replay cache**. The replay cache is the entity that caches the incoming messages with a unique identifier to detect the replay messages.

Process

The Message Replay Detection pattern describes the process of preventing replay attacks at a high level. This implementation pattern provides a more detailed description of that process that is specific to this implementation.

Figure 5.2 illustrates the process to validate messages against a replay cache.

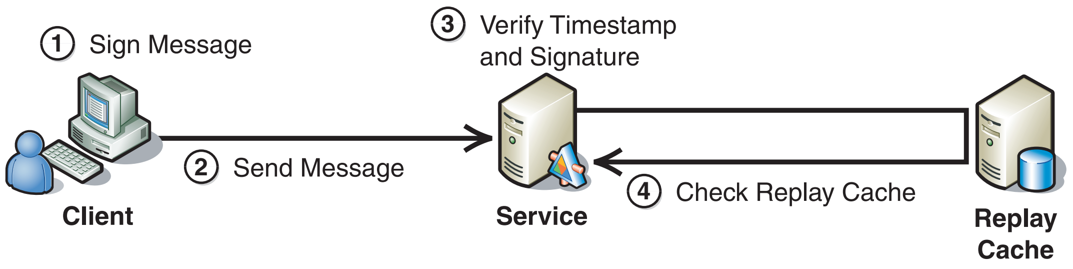


Figure 5.2
The message replay detection process

The process uses the following steps:

1. **The client signs the message**. The client includes a timestamp in the message header and signs the message using a WSE 3.0 policy assertion to provide data origin authentication.
2. **The client sends the message to the service**.
3. **The service verifies the client's signature and the message timestamp**. The service verifies the freshness of the message by checking the message timestamp. If, after accounting for an acceptable clock skew between the client and service, the message timestamp is older than the server will accept, or the timestamp indicates a future time, the message is rejected. If the message timestamp is valid, the message signature is validated. The service then validates the signature on the message to ensure that it came from an expected client, and its content has not been tampered with while in transit.
4. **The service checks the replay cache for the message identifier**. The service checks the replay cache for the message identifier; the message identifier is the contents of the **<SignatureValue>** element in the message signature. If the message identifier is already in the cache, the message is rejected as a duplicate. If the message identifier is not in the cache, the message identifier and cache expiration time for the message are added to the cache.

Implementation Approach

This section provides you with procedures to implement this pattern. The section is divided into the following thee major tasks:

1. **General setup**. This includes a list of steps that apply to all applications for this pattern.
2. **Configure the client**. This includes a list of steps required to configure policy and code on the client.
3. **Configure the service**. This includes a list of steps required to configure policy and code on the service.

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

General Setup

You must install WSE 3.0 on the computers that you use to develop WSE-enabled applications. After you install WSE 3.0, you must enable the client and the service to support WSE 3.0.

► **To enable a Visual Studio project to support WSE 3.0**

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable this project for Web Services Enhancements** check box, and then click **OK**.

Configure the Client

The client requires no special configuration for message replay detection, but it must meet the following requirements:

- You must enable it to use WSE 3.0 and communicate with a WSE 3.0–enabled service as described in the section, "General Setup."
- It must sign the message, and include the message body, addressing headers, and timestamp in the signature.

You also should consider other security requirements for authentication and securing the communication channel. For more information about authentication and securing the communication channel, see the following patterns:

- Direct Authentication in Chapter 1, "Authentication Patterns"
- Brokered Authentication in Chapter 1, "Authentication Patterns"
- Data Confidentiality in Chapter 2, "Message Protection Patterns"
- Data Origin Authentication in Chapter 2, "Message Protection Patterns"

Configure the Service

This section describes the steps required to configure the service and provides example code that you can use to implement message replay detection.

The custom policy assertion for message replay detection requires that an XML signature is present in request messages. When policy is also used on the service to require and verify XML signatures on incoming request messages, that policy should be configured before the message replay detection custom policy assertion is configured. For more information about configuring policy to verify XML signatures on the service, see one of the following implementation patterns in Chapter 3, "Implementing Transport and Message Layer Security":

- Implementing Direct Authentication with UsernameToken in WSE 3.0
- Implementing Message Layer Security with Kerberos in WSE 3.0
- Implementing Message Layer Security with X.509 Certificates in WSE 3.0

If you are not using policy to implement authentication or other forms of message protection for your service, you must first add a text file for the policy cache to your service project in Visual Studio 2005.

► **To add a policy cache file to the service project in Visual Studio**

1. In Visual Studio 2005, right-click the application project, and then click **Add New Item.**
2. Click **Text File**.
3. In the **Name** field, type a name for the file, such as **wse3policyCache.config**.
4. Click **Add**.

Service Policy

The following code example is an example of the configuration for the custom replay detection policy assertion on the service.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
   <extensions>
   ...
      <extension name="replayDetection"
type="Microsoft.Practices.WSSP.WSE3.QuickStart.ReplayDetection.CustomAssertions.Re
playDetectionAssertion,
Microsoft.Practices.WSSP.WSE3.QuickStart.ReplayDetection.CustomAssertions"/>
   </extensions>
   <policy name="ReplayDetectionService">
<replayDetection cacheLifetimeInSeconds="1200" maxMessageAgeInSeconds="600" />
...
   </policy>
</policies>
```

The **replayDetection** assertion has the following two configurable parameters:

- *cacheLifetimeInSeconds*. This parameter specifies how long in seconds identifiers will remain in the replay cache. In the preceding example, this parameter is configured for 1,200 seconds or 20 minutes.
- *maxMessageAgeInSeconds*. This parameter specifies the maximum message age in seconds that is tolerated by the assertion without accounting for clock skew. In the preceding example, this parameter is configured for 600 seconds or 10 minutes.

Paste the **<extension>** and **ReplayDetectionService** policy elements from the example into your policy configuration file.

Note: If you are pasting into a pre-existing policy file, you may also have to add the opening and closing **<extensions>** elements around the **<extension>** element.

The order that you use to place the replay detection assertion within your policy only matters relative to the other policy assertions that you may use. For example, if you are doing message validation in a custom policy, place this assertion after the message replay detection assertion. If you have multiple security assertions defined in policy, you should place this assertion before each security assertion. Security assertions are those assertions that are used to sign and encrypt messages; they include all the WSE 3.0 turnkey policy assertions, with the exception of the **usernameOverTransportSecurity** turnkey assertion. For more information about the message validation custom assertion, see Implementing Message Validation in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."

If you are not using policy to implement authentication or message protection for your service as described earlier in this section, you will need to enable policy support by directly modifying the service's Web.config file because WSE does not recognize custom policy assertions when it parses the policy cache file; it disables policy support if you attempt to configure it using the WSE Settings tool. If you have to enable policy support after a custom policy assertion has been added to your policy cache, you have to add a **<policy>** element to the service's Web.config file to enable policy support, as shown here.

```
<microsoft.web.services3>
...
   <policy fileName="wse3policyCache.config" />
...
</microsoft.web.services3>
```

Replace the value specified for the **fileName** attribute with the file path and name of your policy cache file.

WSE 3.0 also has an important setting in this context, **<timeToleranceInSeconds>**. This setting corresponds to the acceptable time difference (clock skew) between the sender and the recipient of a message. The **<timeToleranceInSeconds>** setting is configured to 300 seconds or 5 minutes by default. However, you can change this value in the service's Web.config file if you require a different value.

Note: The **<timeToleranceInSeconds>** setting is shared, so changing it may also affect security token managers and other policy assertions operating in the same virtual directory as the service.

The following example code configuration snippet provides an example of this setting in the service's Web.config file. Note that in the example, the value is set to the default value 300 seconds.

```
<microsoft.web.services3>
...
<security>
<timeToleranceInSeconds value="300" />
...
</security>
</microsoft.web.services3>
```

A message is accepted or rejected according to logic that takes into consideration the potential time difference between the sender and receiver and an acceptable age for the message to account for longer delays in message transport (for example, in store and forward scenarios). The following logic is applied when determining whether to accept an incoming message:

1. The server calculates the message age by subtracting the created value on the message from the current server time. Because of clock skew between the sender and recipient computers, this value can be positive or negative. If the result of this calculation is greater than zero, the message appears to have been created in the past; if the value is less than zero, it appears to have been created in the future.
2. For a message that appears to have been created in the past or if the server and message creation times are identical, the message will be accepted only when its message age is less than or equal to the values for the **maxMessageAgeInSeconds** parameter plus the **<timeToleranceInSeconds>** setting,
3. For a message that appears to have been created in the future (where the message age is a negative value), the Maximum Message Age setting is not considered, because any delay in message transmission would already have made the message age closer to zero. Instead, the mathematical absolute value of the message age is used. If this value is less than or equal to the Time Tolerance setting, the message is accepted.

Messages are held in the cache for at least as long as the value that is defined in the **CacheLifetimeInSeconds** setting. To ensure that the server cannot accept a message after a duplicate message has been removed from the cache, the **CacheLifetimeInSeconds** setting must be set to at least the Maximum Message Age + Time Tolerance*2.

Figure 5.3 illustrates the relationship between the previously described configuration settings.

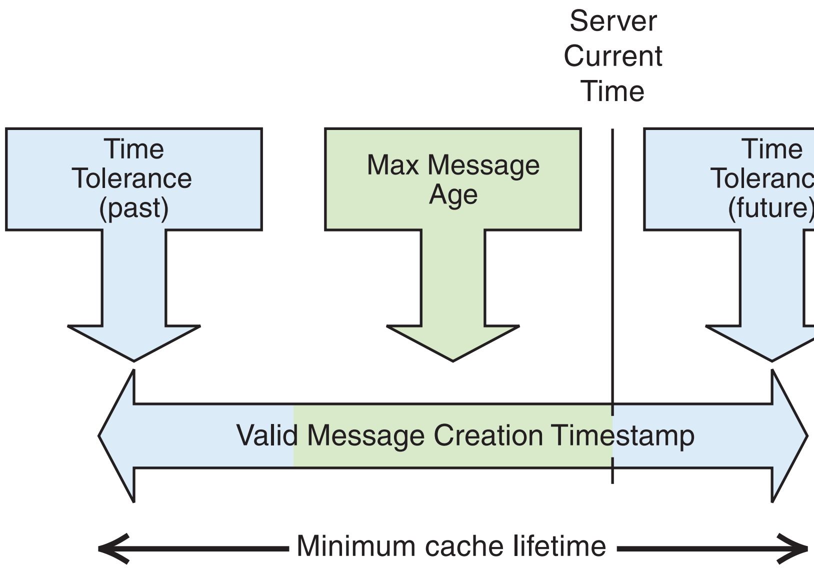


Figure 5.3
The relationship between the configuration settings

In the example code, the following setting values are configured:

- **<timeToleranceInSeconds>**. This value is set to the default of 300 seconds or 5 minutes.
- **maxMessageAgeInSeconds**. This value is set to 600 seconds or 10 minutes.
- **cacheLifetimeInSeconds**. This value is set to 1,200 seconds or 20 minutes.

These configuration settings are valid because message age plus twice the time tolerance or (600 + (300 x2)) does not exceed the configured cache lifetime of 1,200 seconds.

To bind the policy assertion to your Web service, add the following attribute before the class declaration in your Web service code.

```
[Policy("ReplayDetectionService")]
```

Replay Detection Custom Policy Assertion Code

The following code example displays the message replay detection custom policy assertion.

```
using System;
using System.Xml;
using System.Collections.Generic;

using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
using Microsoft.Web.Services3.Design;
using Microsoft.Web.Services3.Configuration;

namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.ReplayDetection.CustomAssertions
{
    public class ReplayDetectionAssertion : PolicyAssertion
    {
        #region Custom Fields
        private int cacheLifetime;
        private int maxMessageAge;
        #endregion

        #region PolicyAssertion Methods
        public override SoapFilter CreateClientInputFilter(FilterCreationContext
context)
        {
            return null;
        }

        public override SoapFilter CreateClientOutputFilter(FilterCreationContext
context)
        {
            return null;
        }

        public override SoapFilter CreateServiceInputFilter(FilterCreationContext
context)
        {
            return new ReplayDetectionAssertion.ServiceInputFilter(this);
        }
```

(continued)

(continued)

```
        public override SoapFilter CreateServiceOutputFilter(FilterCreationContext
context)
        {
            return null;
        }
        public override void ReadXml(System.Xml.XmlReader reader,
IDictionary<string, Type> extensions)
        {
            if (reader == null)
                throw new ArgumentNullException("reader");
            if (extensions == null)
                throw new ArgumentNullException("extensions");

            bool isEmpty = reader.IsEmptyElement;

            string cacheLifetime = reader.GetAttribute("cacheLifetimeInSeconds");
            if (!string.IsNullOrEmpty(cacheLifetime))
            {
                try
                {
                    this.cacheLifetime = Math.Abs(int.Parse(cacheLifetime));
                }
                catch
                {
                    throw new FormatException(Messages.CacheLifetimeFormat);
                }
            }
            else
            {
                this.cacheLifetime = -1;
            }

            string maxMessageAge = reader.GetAttribute("maxMessageAgeInSeconds");
            if (!string.IsNullOrEmpty(maxMessageAge))
            {
                try
                {
                    this.maxMessageAge = Math.Abs(int.Parse(maxMessageAge));
                }
                catch
                {
                    throw new FormatException(Messages.MaxMessageAgeFormat);
                }
            }
            else
            {
                this.maxMessageAge = -1;
            }
```

(continued)

(continued)

```
            reader.ReadStartElement("replayDetection");
            if (!isEmpty)
            {
                reader.ReadEndElement();
            }
        }
        public override void WriteXml(System.Xml.XmlWriter writer)
        {
            writer.WriteStartElement("replayDetection");

            if (this.cacheLifetime != -1)
                writer.WriteAttributeString("cacheLifetimeInSeconds",
this.cacheLifetime.ToString(System.Globalization.CultureInfo.InvariantCulture));

            if (this.maxMessageAge != -1)
                writer.WriteAttributeString("maxMessageAgeInSeconds",
this.maxMessageAge.ToString(System.Globalization.CultureInfo.InvariantCulture));

            writer.WriteEndElement();
        }
        #endregion

        #region Custom SoapFilters
        protected class ServiceInputFilter : SoapFilter
        {
            #region Custom Fields

            private int cacheLifetime;
            private int maxMessageAge;
            #endregion

            #region Constructors
            public ServiceInputFilter(ReplayDetectionAssertion assertion)
                : base()
            {
                this.cacheLifetime = assertion.cacheLifetime;
                this.maxMessageAge = assertion.maxMessageAge;
            }
            #endregion

            #region ReceiveSecurityFilter Methods
            public override SoapFilterResult ProcessMessage(SoapEnvelope envelope)
            {
                DetectReplayedMessage(envelope);
                return SoapFilterResult.Continue;
            }

            private void DetectReplayedMessage(SoapEnvelope envelope)
            {
                CheckMessageAge(envelope);
```

(continued)

(continued)

```
                // Calculate the message expiration time based on the cache
lifetime configured in the policy assertion.
                //Gets the current time in UTC.
                // UTC is used for two reasons:
                // 1) Daylight savings is not applied to UTC. If the local server
clock accounts for daylight savings,
                // the server hosting the cache would prematurely delete data from
the cache when the clock is rolled forward in the spring;
                // this allows a window for replay detection of approx 40 minutes
based on our default replay settings.
                // 2) UTC provides a common time reference if the Web service and
database server are in different time zones.
                DateTime messageExpirationDate =
DateTime.Now.AddSeconds(this.cacheLifetime).ToUniversalTime();

                foreach (ISecurityElement element in
envelope.Context.Security.Elements)
                {
                    if (element is MessageSignature)
                    {
                        MessageSignature signature = (MessageSignature)element;

                        string messageKey =
Convert.ToBase64String(signature.Signature.SignatureValue);

                        // Add the message to the cache.
                        CacheHelper.Cache(messageKey, messageExpirationDate);
                    }

                }
            }
            #endregion

            #region Custom Methods

            /// <summary>
            /// Validates the message timestamp to avoid replay attacks.
            /// </summary>
            private void CheckMessageAge(SoapEnvelope envelope)
            {
                // Gets the message timestamp.
                DateTime timestamp = envelope.Context.Security.Timestamp.Created;

                DateTime currentDate = DateTime.Now;
```

(continued)

(continued)

```
                // Computes the time difference between the message timestamp and
the current time.
                TimeSpan timeDifference = currentDate.Subtract(timestamp);

                double messageAgeInSeconds = timeDifference.TotalSeconds;

                // The first condition checks for messages where sender's clock +
network lag is slower than
                // the server's clock because we do not want to consider message
age if the sender's clock
                // is faster.
                // The second condition accounts for messages where the sender's
clock is faster than the server's clock.
                if ((messageAgeInSeconds > this.maxMessageAge +
WebServicesConfiguration.SecurityConfiguration.TimeToleranceInSeconds.TotalSeconds
)
                    || (messageAgeInSeconds < 0 && Math.Abs(messageAgeInSeconds) >
WebServicesConfiguration.SecurityConfiguration.TimeToleranceInSeconds.TotalSeconds
))
                {
                    throw new SecurityFault(Messages.AgeRequirementsNotSatisfied);
                }
            }

            #endregion
        }
        #endregion
    }
}
```

The preceding code example uses a class named **CacheHelper** to abstract the interaction with the message replay cache. As an example, this implementation uses a database for the message replay cache. Based on the requirements for your application and your environment, you may want to implement a different kind of cache. The source code for the **CacheHelper** class is provided in the section, "Replay Cache."

All times are converted to the Universal Time Convention (UTC) in the previous example for two reasons:

- To compensate for when the Web service host and replay cache host are in different time zones.
- To compensate for daylight savings time because it is not applied to the UTC. When the Web service host adjusts its clock an hour forward for daylight savings time, messages are not unintentionally deleted from the cache; if they were unintentionally deleted, there would be an opportunity for message replay.

Replay Cache

The following code example provides an example of a **CacheHelper** class that the policy assertion uses to interact with a database replay cache.

Cache expiration is calculated on the Web service in this pattern to centralize all policy logic on the Web service that is implementing replay detection instead of spreading configuration settings across the Web server and the database server that is hosting the replay cache. This example assumes that there is close clock synchronization between the server hosting the service and the database server. If this assumption is invalid for your environment, make adjustments to this implementation to compensate for a lack of clock synchronization between the server that is hosting the Web service and the database server.

Note: Instead of calculating cache lifetime for the message in the policy assertion, you can set the default value on the message expiration column in the replay cache database table to the current time on the database server, and the cache lifetime when record is inserted. The tradeoff of this approach is that instead of configuring all of your settings on the Web service implementing message replay detection, you must configure and calculate the cache lifetime on the database server.

```
using System;
using System.Configuration;
using System.Data.SqlClient;

namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.ReplayDetection.CustomAssertions
{
    /// <summary>
    /// Provides static methods to manage cache.
    /// </summary>
    class CacheHelper
    {
        private const string ConnectionStringName = "CacheHelper";

        private CacheHelper() { }

        /// <summary>
        /// Adds to cache the provided object indexed by the provided key until
the provided expiration date.
        /// </summary>
        /// <param name="value">Object to be cached</param>
        /// <param name="expirationDate">Object cache's expiration date</param>
        public static void Cache(object value, DateTime expirationDate)
        {
            string connectionString = GetConnectionString();
```

(continued)

(continued)

```
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand command = new
SqlCommand("usp_AddMessageToCache", connection))
                {
                    command.CommandType = System.Data.CommandType.StoredProcedure;
                    command.Parameters.Add("@messageIdentifier",
System.Data.SqlDbType.VarChar, 200);
                    command.Parameters.Add("@expirationTime",
System.Data.SqlDbType.DateTime);

                    command.Parameters["@messageIdentifier"].Value = value;
                    command.Parameters["@expirationTime"].Value = expirationDate;

                    try
                    {
                        int rowsUpdated = command.ExecuteNonQuery();

                        // No row was updated because a duplicate key was
detected.
                        if (rowsUpdated == -1)
                        {
                            throw new
InvalidOperationException(Messages.ExistentItem);
                        }
                    }
                    catch (SqlException sqlException)
                    {
                        // Check for the SQL error 2601 because this error means
"Duplicate key" and a friendly error
                        // message is returned in that case.
                        if (sqlException.Number == 2601)
                        {
                            throw new
InvalidOperationException(Messages.ExistentItem);
                        }
                        else
                        {
                            throw;
                        }
                    }

                    connection.Close();
                 }
            }

        }
```

(continued)

(continued)

```
        /// <summary>
        /// Gets the configured connection string from the configuration system.
        /// </summary>
        /// <returns></returns>
        private static string GetConnectionString()
        {
            ConnectionStringSettings settings =
ConfigurationManager.ConnectionStrings[ConnectionStringName];

            if(settings == null)
                throw new
ConfigurationErrorsException(String.Format(Messages.ConnectionStringNotConfigured,
ConnectionStringName));

            if (String.IsNullOrEmpty(settings.ConnectionString))
                throw new
ConfigurationErrorsException(String.Format(Messages.ConnectionStringNotConfigured,
ConnectionStringName));

            return settings.ConnectionString;
        }
    }
}
```

In the preceding code example, thrown exceptions accept a defined value in their constructors for the exception message parameter as defined by a **Messages** object, such as the **Messages.ConnectionStringNotConfigured** value. These values refer to resource strings that provide a message for the exceptions that are thrown. Substitute these as appropriate with a simple exception message to provide information about why the exception is being thrown.

In the previous example, the **CacheHelper** class requires a connection string named "CacheHelper" in the application's configuration file. This connection string provides connection information for the database where the replay cache is hosted.

```
...
<connectionStrings>
<add name="CacheHelper" connectionString="Data Source=localhost;Integrated
Security=SSPI;Initial Catalog=ReplayDetection;"/>
</connectionStrings>
...
```

In this implementation, the database cache resides on a computer running SQL Server. Using SQL Server provides the following benefits:

- **It supports a Web farm scenario**. You can easily share the database cache between servers in a Web farm and the software supports concurrent access to the cache.
- **It provides cache stability**. In-memory caches are cleared after a certain period of inactivity on the server, after periodic recycling of application process threads, or after you restart the server. A SQL database provides data consistency for the cache, regardless of the state of the Web service application process or its threads.

Using a database as a replay cache also has disadvantages:

- **Performance**. Because databases store data on disks, data retrieval and updates are slow in comparison to in-memory caching.
- **Connectivity**. If a database cache is hosted on a remote server, the cache mechanism will not function if the policy assertion cannot connect to the database server.

You can take several steps to optimize the performance of the database cache, including:

- **In-memory tables**. The database server can be configured to keep the replay cache table resident in memory instead of reading and writing from the physical storage media.
- **Index tuning and optimization**. Operations on the database table can be optimized by tuning the indexes on the table.

For more information about SQL Server performance optimization, see Optimizing Database Performance Overview.

The SQL Server replay cache in this example consists of a table, two stored procedures, and a SQL Server Agent job. The replay cache database table is named ReplayCache. The structure for the replay cache database table is summarized in Table 5.1.

Table 5.1: Replay Cache Database Structure Summary

Name	Type	Description	Notes and constraints
MessageID	Integer	Identity column.	Primary key.
MessageIdentifier	varchar(200)	Message identifier.	Unique, required. You may have to increase the column width to account for longer message signature values.
ExpirationTime	Datetime	Time when the message expires in the cache.	Required.

The following code example displays a SQL script that you can use to create the table and indexes.

```
CREATE TABLE [dbo].[ReplayCache] (
    [ReplayCacheID] [int] IDENTITY (1, 1) NOT NULL ,
    [MessageIdentifier] [varchar] (200) ,
    [ExpirationTime] [datetime] NOT NULL
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[ReplayCache] WITH NOCHECK ADD
    CONSTRAINT [PK_ReplayCache] PRIMARY KEY CLUSTERED
    (
        [ReplayCacheID]
    ) ON [PRIMARY]
GO
ALTER TABLE [dbo].[ReplayCache] ADD
    CONSTRAINT [DF_ReplayCache_ExpirationTime] DEFAULT (getdate()) FOR
[ExpirationTime]
GO
 CREATE UNIQUE INDEX [IX_MessageIdentifier] ON
[dbo].[ReplayCache]([MessageIdentifier]) WITH PAD_INDEX ON [PRIMARY]
GO
 CREATE INDEX [IX_ExpirationTime] ON [dbo].[ReplayCache]([ExpirationTime]) WITH
PAD_INDEX ON [PRIMARY]
GO
```

You will probably have to modify the preceding script and optimize the indexes to suit your needs or run the script and tune the indexes using the tools available with SQL Server.

The two stored procedures for the replay cache are:

- **usp_AddMessageToCache**. This stored procedure inserts the message identifier and expiration time calculated in the policy assertion into the replay cache database table. Because the cache will experience a lot of concurrent activity, check for a SQL error code of 2601 after this stored procedure executes. If a SQL error does occur with a return code of 2601, the unique constraint on the message identifier column has been violated. This means that between the time the policy assertion checked to determine if the message identifier was already in the cache and the time it attempted to insert it, another process already inserted the message identifier into the cache. This situation is treated as a replay attempt.
- **usp_ClearExpiredMessages**. This stored procedure is executed by the SQL Server Agent job, which is described in the next section, to remove expired messages from the cache.

The following script creates the two stored procedures for the replay cache.

```
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS OFF
GO
CREATE PROCEDURE [dbo].[usp_AddMessageToCache] (@messageIdentifier varchar(200),
@expirationTime datetime) AS
INSERT INTO ReplayCache (MessageIdentifier, ExpirationTime)
VALUES (@messageIdentifier, @expirationTime);
GO
CREATE PROCEDURE [dbo].[usp_ClearExpiredMessages] AS
DELETE FROM ReplayCache
WHERE ExpirationTime <= GETUTCDATE();
GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO
```

After you run the preceding script, make sure to grant execute permissions on **usp_AddMessageToCache** to the service account that the service runs under. It is important to exercise the best practice of minimum privilege on the database table. Grant permissions only to execute the stored procedures to the service account under which the Web service implementing replay detection runs. Do not allow the Web service to directly modify data in the database table. Also, make sure that the communication between the service implementing replay detection and the database cache is secure.

For more information about security best practices for SQL Server 2000, see SQL Server 2000 SP3 Security Features and Best Practices.

Cache Cleanup

You must clear the cache at regular intervals to regulate its size. A SQL Server Agent job clears the database cache. The job is scheduled to execute the **usp_ClearExpiredMessages** stored procedure at approximately the same interval as the cache lifetime value configured in the replay detection policy assertion. For example, in this pattern, the cache lifetime is configured at 20 minutes in the policy assertion. The SQL Server Agent job executes every 22 minutes to keep the cache reasonably clear.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The implementation provides a solution to prevent the service from processing replayed messages. It does this by rejecting messages that the service has previously received within the valid processing time for them.

Liabilities

The liabilities associated with the Implementing Message Replay Detection in WSE 3.0 pattern include the following:

- There is a small probability that the **SignatureValue** of two different messages could be the same. This would result in one of the messages getting falsely rejected as a replay attempt. The probability for this to occur is very small based on the number of value combinations that could make a SignatureValue, but it remains possible.
- This pattern describes how to perform replay detection using WS-Security. When you use it in conjunction with other protocols, such as reliable messaging, there is a possibility that resent messages could be falsely rejected as replay attempts. You may have to modify the approach to message replay detection described in this pattern to use values in the message that distinguish a resent message from the original message. For more information about reliable messaging, see Reliable Message Delivery in a Web Services World: A Proposed Architecture and Roadmap on MSDN.
- It may be difficult to find an effective replay cache mechanism that meets all of the requirements for the implementation. Consider the following requirements before choosing the replay cache mechanism to implement:
  - To operate on a Web farm, you must share it across multiple servers.
  - It must support frequent and concurrent updates in real time.
  - Cache performance is very important. If you are using a database, it is likely that you will have to optimize the database to function as a replay cache. In-memory caches perform best, but they depend on the life cycle of the application processes and they do not provide cache data consistency.

Note: An alternative approach to implementing a message replay detection assertion is to extend an existing WSE 3.0 turnkey assertion to provide message replay detection capability. For example, the **MutualCertificate11Assertion** turnkey assertion class and its security filters can be extended to provide message replay detection capabilities immediately after the signature is verified by the receive security filter in the assertion.

Security Considerations

Security considerations associated with the Implementing Message Replay Detection in WSE 3.0 pattern include the following:

- You must set the cache lifetime for the custom policy assertion for a longer time than the maximum message age configured in the policy assertion added to twice the WSE 3.0 configuration value for time tolerance in seconds. This should not depend on the expiration of the message specified by the sender.
- Replay caches do not inherently provide a means for a service to detect cache tampering. If replay cache tampering is an identified threat that you choose to mitigate, as revealed by a proper threat analysis of your application, consider requiring the service (or services on a Web farm) to create a Hashed Message Authentication Code (HMAC) or digital signature on the cache contents to verify the cache's integrity. This approach is effective to mitigate cache tampering, but it also degrades the performance of the replay detection mechanism.
- For simplicity, the examples in this pattern do not apply mitigation techniques against all possible threats. For example, all input should be validated. For more information, see Message Validator in Chapter 5, "Service Boundary Protection Patterns."
- If you are using a perimeter service router to route the same types of messages to several different service endpoints, you have to make sure that a service will not process a replayed message that was already processed by one of the other service endpoints that receives messages from the router. To mitigate a message replay across multiple services, you must either make sure that the replay cache is shared by all the services or implement message replay detection on the perimeter service router.

Message Validator

Context

A Web service interacts with other applications over a network. Incoming data may be malformed and may have been transmitted for malicious purposes. There is also a risk of injection attacks, where data from incoming messages is tampered with to include additional syntax.

Problem

How do you protect Web services from malformed or malicious content?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Malicious content poses a risk to the Web service**. An attacker can insert syntax in a request message to cause the Web service or other downstream systems that process the received data to behave in an undesirable manner. The attacker can do this through injection attacks, such as XML injection, SQL injection, or HTML/client script injection. Web services that do not require access control are especially susceptible because they have no means to limit to a smaller, more trusted group, the number of clients that can access them.
- **There is a risk of attackers bypassing client validation techniques by using alternative clients or by modifying data after it has left the client**. Web services must be designed to be autonomous and perform their own input validation instead of trusting the validation that is performed in the client application.

The following condition is an additional reason to use the solution:

- **An attacker can use malformed or oversized messages to launch a denial-of-service attack**. Denial of service attacks can take advantage of the multiplier effect, where a malformed or oversized message causes a disproportionate increase in the use of resources, such as a server's CPU time, memory usage, or database connections.

Solution

Assume that all input data is malicious until proven otherwise, and use message validation to protect against input attacks, such as SQL injection, buffer overflows, and other types of attacks. The message validation logic enforces a well-defined policy that specifies which parts of a request message are required for the Web service to successfully process it. It validates the XML message payloads against an XML schema (XSD) to ensure that they are well-formed and consistent with what the Web service expects to process. The validation logic also measures the messages against certain criteria by examining the message size, the message content, and the character sets that are used. Any message that does not meet the criteria is rejected.

Participants

Message validation involves the following participants:

- **Client**. The client accesses the Web service.
- **Service**. The service is the Web service that processes requests received from clients. The service implements the message validation logic.

Process

Figure 5.4 illustrates the process that is used by message validation logic to intercept request messages and verify that they are acceptable for processing by the service.

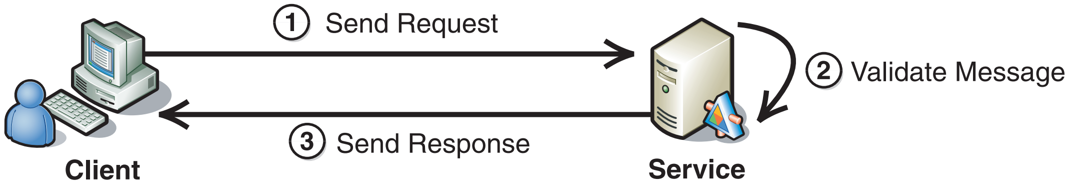


Figure 5.4
Message validation occurring at a Web service

As illustrated in Figure 5.4, the process for message validation is described in the following steps:

1. **The client sends a request message to the service**. The validation process itself is hidden from the client.
2. **The service validates the message**. The message validation logic makes a number of checks to validate the message. Checks can include:
   - Comparing the size of the request against the maximum allowable size that is specified for request messages.
   - If the message is signed, verifying the signature to ensure that the message has not been tampered with in transit.
   - Verifying that the message payload is well-formed and conforms to a predefined schema, with acceptable data types and ranges of values.
   - Parsing the entire request message for malicious content. Potentially, malicious content can be placed in either the SOAP message elements or in the message payload, so both are checked.
3. **The service processes the request and responds to the client**. If the request passes all the validation checks that are performed by the message validator, the service processes the message and may issue a response to the client.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of the Message Validator pattern include the following:

- The Web service is protected from malformed and malicious content. This helps protect against injection attacks, even for Web services that do not implement access control.
- The Web service performs validation independently of the client — it does not accept messages simply because they have already been validated by the client.

Liabilities

The liabilities associated with the Message Validator pattern include the following:

- Message validation logic does not process binary message content, such as attachments. For message validation logic to process binary attachments, it needs to be capable of recognizing each type of binary attachment that it encounters to ensure that they are free of malicious content. Specifying a maximum message size helps to protect against injection attacks in binary attachments. However, validation of binary data should be handled by antivirus filters.
- If a message is encrypted with message layer security, it may not be possible to inspect data for malicious content unless the message is decrypted beforehand or the validation logic has access to the decryption key.
- If data is protected by transport layer security, the entire channel is encrypted and decrypted at end points. As a result, message validation cannot occur at any intermediaries between those points.

Security Considerations

Security considerations associated with the Message Validator pattern include the following:

- Message validation can help protect against denial of service attacks, but the message validation logic must be very efficient when it conducts its validation checks. Otherwise, the message validation logic may be a system bottleneck and may itself become the target of a denial of service attack. Malformed content can include very large messages, in some cases for the purposes of launching a denial of service attack. You should make the maximum message size large enough to allow legitimate messages to be accepted but small enough to prevent attacks.
- Using a validating parser and verifying the input message against its XML Schema (XSD) result in a significant increase in CPU processing. And, even though XML Schema (XSD) has the capability to specify data range validations and it supports the use of regular expressions, many schemas use data types, such as string, which do not prevent many forms of injection attacks.

- Instead of building the message validation logic into the Web service itself, you can place it in an intermediary. This allows several Web services to use the same intermediary, and it enables each Web service to dedicate its resources to processing legitimate messages. It also ensures that invalid messages never reach the Web service. However, using an intermediary in this way can create a single point of failure, which may become a target of attack.
- XML message payloads that contain a CDATA field can be used to inject illegal characters that are ignored by the XML parser. If CDATA fields are necessary, you must inspect them for malicious content.
- The Web service may obtain data for response messages from external sources. There is no guarantee that external data sources properly validate data. Passing responses without message validation makes the Web service a potential "carrier" of malicious input from external data sources. You should consider filtering Web service response messages that are returned to the client.

Related Patterns

The following child pattern is related to the Message Validation pattern:

- **Implementing Message Validation in WSE 3.0**. This pattern provides steps and recommendations to implement message validation at the message layer with WSE 3.0.

Implementing Message Validation in WSE 3.0

Context

You are implementing a Web service that uses Web Service Enhancements (WSE) 3.0. The Web service must validate request messages received from clients to make sure that they are not malformed and do not contain malicious content.

Objectives

This implementation of the Message Validator pattern has the following objectives:

- Prevent the service from processing request messages that are larger than a specified size.
- Prevent the service from processing messages that are not well-formed or that do not conform to an expected XML schema.
- Validate input messages before deserializing them into .NET data types so that they can be interpreted as regular expressions.
- Demonstrate how to use WSE 3.0 custom assertion to implement message validation.
- Use ASP.NET and WSE 3.0 configuration settings to limit usage of system resources such as CPU.

Content

This implementation pattern includes the following sections:

- **Implementation Strategy**. This section provides a high-level description of the strategy used to implement the Message Validation pattern.
- **Implementation Approach**. This section describes the steps required to implement this pattern:
  - Configure the client
  - Configure the service
- **Resulting Context**. This section outlines the benefits, liabilities, and other considerations related to the pattern,

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

To implement message validation on a Web service, you use a combination of application configuration, code implementation, and filtering in WSE 3.0. Use one or more of the following methods to perform message validation:

- Set the maximum request size in the service's configuration file to limit the size of messages that the service will process.
- Validate each incoming request message to ensure that it is well-formed XML, that it contains all of the parts required by the service, and that the contents of the message conforms to an expected structure as defined by an XML Schema (XSD).
- Use regular expression checking to ensure that input contains only valid data and does not contain malicious SQL, HTML, or JavaScript code that could lead to code injection attacks.
- Use regular expressions to ensure that complex data types (such as social security numbers and telephone numbers) are received in a format that the service can process.

Note: You should conduct a thorough threat analysis of your service application to determine where in the code you should perform message validation and to determine which methods of message validation you should use.

To fully understand this pattern, you must have some experience with the .NET Framework, WSE 3.0, and Web service development.

Participants

This implementation pattern requires the following participants:

- **Client**. The client accesses the Web service.
- **Service**. The service is the Web service that processes requests received from clients. The service implements the message validation logic.

Process

The Message Validator pattern describes the message validation process at a high level. This implementation pattern provides a refined description of that process specific to the WSE 3.0 implementation.

Figure 5.5 illustrates the process by which message validation logic intercepts request messages and verifies that they are acceptable for processing by the service.

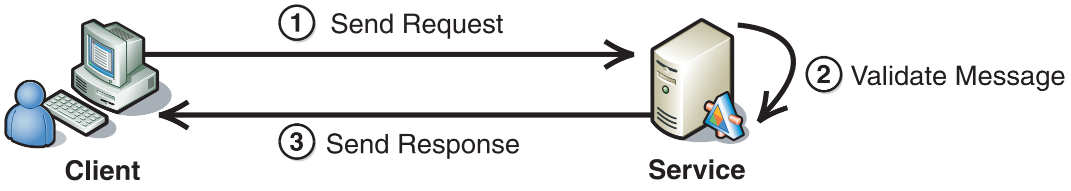


Figure 5.5
Validating a request message

The process uses the following steps:

1. The client sends a request message to the service.
2. **The service validates the message**. The service uses a number of different validation checks to prevent malicious input. These include:
   - Comparing the size of the request to the value established for the **maxRequestLength** attribute of the **<httpRuntime>** element in the application's configuration file, which is specified in kilobytes. **maxRequestLength** specifies the maximum allowable size for request messages. If the message exceeds this value, the service does not process the message, and it returns an error.

Note: You can set other values in the **<httpRuntime>** element to control response, resource usage for handling requests, and timeouts. For more information about **<httpRuntime>**, see <httpRuntime> Element in the *.NET Framework General Reference* on MSDN.

- Checking the format of the request message to ensure that the message is formed correctly and that all of the required message parts are present. The service uses WSE policy assertions to make sure that all required message parts are present. The service can use the **requireActionHeader** policy assertion to verify that the message contains a WS-Addressing action header. The service can use the **requireSoapHeader** policy assertion to verify that the message contains other SOAP header elements, such as an addressing header and a message ID. For more information about WSE 3.0 policy assertions, see Policy Assertions in the WSE 3.0 product documentation on MSDN.
- Verifying that the XML in the message payload is well-formed and that it conforms to a predefined schema with acceptable data types and ranges of values. The service uses an XML Schema (XSD) to validate the contents of the message body. If a specific schema is not required for validation, it can use an XML parser to validate the request body. The service can use an XML Schema (XSD) to perform structural validation, data type validation, cardinality of child elements to parent elements, numeric value ranges, and regular expression validation for character patterns and ranges.
- Parsing the request message for malicious content. The service can use regular expressions to ensure that the messages contain only valid data. Regular expression validation can be implemented either in the XML Schema (XSD) or in code. Also, the service can use parameterized SQL queries to access and modify data in databases to mitigate the risk of SQL injection.

3. **The service processes the request and responds to the client**. If the request passes all validation checks performed by the message validator, the service processes the message.

Implementation Approach

This section describes how to implement the pattern. The section is broken into two major tasks:

- **Configure the client**. This section describes the steps required to configure policy and code for the client.
- **Configure the service**. This section describes the steps required to configure policy and code for the service.

This pattern does not specifically address other security requirements for authentication and securing the communication channel. For more information about authentication and securing the communication channel, see the following patterns:

- Direct Authentication in Chapter 1, "Authentication Patterns."
- Brokered Authentication in Chapter 1, "Authentication Patterns."
- Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Data Origin Authentication in Chapter 2, "Message Protection Patterns."

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

Configure the Client

The client requires no special configuration for message validation. The client should be able to recognize and properly handle validation exceptions thrown by the service.

Configure the Service

If you use policy to implement authentication and message protection for your service, you should configure it before you attempt to use the custom policy assertion provided in this implementation. For policy-based authentication and message protection examples, see one of the following implementation patterns in Chapter 3, "Implementing Transport and Message Layer Security":

- Implementing Message Layer Security with X.509 Certificates in WSE 3.0
- Implementing Message Layer Security with Kerberos in WSE 3.0
- Implementing Direct Authentication with UsernameToken in WSE 3.0

If you do not use policy to implement authentication and/or message protection for your service, you must enable support for WSE 3.0 and add a text file for the policy cache to your service project in Visual Studio 2005 before using the custom policy assertion provided in this pattern.

► To enable the service project to support WSE 3.0

1. In Visual Studio 2005, right-click the application project, and then click **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable this project for Web Services Enhancements** check box, select the **Enable Microsoft Web Services Enhancement SOAP Protocol Factory** check box, and then click **OK**.

► To add a policy cache file to the service project in Visual Studio

1. In Visual Studio, right-click the application project, and then click Add New Item**.**
2. Click **Text File**.
3. In the **Name** field, type a name for the file, such as **wse3policyCache.config**.
4. Click **Add**.

This section is divided into subsections; each subsection describes a message validation technique. You do not always have to implement all the message validation techniques. You should complete a thorough threat analysis of the service to determine which techniques to use.

Whether you implement some or all of the message validation techniques, you should implement them in the order that they are described. The order in which the message validation techniques occur depends on where they are implemented in the platform. In this pattern, the request size must be checked before any other step. The custom policy assertion must be applied in the pipeline after the message is decrypted but before the request is processed by the service. Regular expression checking, if implemented in the XML Schema (XSD), occurs when the request is validated against the message schema in the policy assertion. Otherwise, regular expression checking occurs where the code is implemented, most likely in the service code. Parameterization of SQL queries occurs when the query is created, prior to execution on the database server.

The point at which the body validator assertion is specified does not matter relative to other assertions defined to protect the message, because decryption and signature verification is applied further up the communication pipeline from assertions applied for message validation.

Configure Maximum Request Length

To limit the size (in kilobytes) of messages that the service will process, you should specify a value for the **maxRequestLength** attribute of the **<httpRuntime>** element in the service's Web.config file. This value should be set according to the largest request message that you can reasonably expect the service to process. If you do not specify a value for this setting, the default value is 4096 KB. The following XML example shows a maximum request length set to 300 KB.

```
<configuration>
      ...
  <system.web>
     <httpRuntime maxRequestLength="300"/>
      ...
  </system.web>
   ...
</configuration>
```

If your service uses a protocol other than HTTP (such as TCP), the WSE **<maxMessageLength>** setting can be used to limit the size (in kilobytes) of incoming requests, assuming that you are using the **SoapClient/SoapService** model for your service. The default value for the **length** attribute of the **<maxMessageLength>** element is 4096 KB. The following configuration example shows the **<MaxMessageLength>** set to 1024 KB for a service that uses the **SoapClient/SoapService** model.

```
<configuration>
...
  <microsoft.web.services3>
  ...
    <messaging>
      <maxMessageLength value="1024" />
    </messaging>
    ...
  </microsoft.web.services3>
  ...
</configuration>
```

For more information about using the **SoapClient/SoapService** classes for messaging, see How To: Send and Receive a SOAP Message by Using the **SoapClient** and **SoapService** Classes in the WSE 3.0 product documentation on MSDN.

Required Message Part/Schema Validation

This implementation pattern uses policy assertions to check for required message parts and to validate the message schema. The following example policy file provides an example of policy assertions for the service. Other polices that would be present to sign, encrypt, and provide authentication capabilities have been omitted for brevity.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
 <extensions>
 ...

 <extension name="bodyValidator"
type="Microsoft.Practices.WSSP.WSE3.QuickStart.MessageValidation.CustomAssertions.
BodyValidatorAssertion,
Microsoft.Practices.WSSP.WSE3.QuickStart.MessageValidation.CustomAssertions"/>
 </extensions>
 <policy name="MessageValidationService">
       <bodyValidator xsdPath="Configuration\GetCustomers.xsd" />
       ...
       <requireSoapHeader name="MessageID"
namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
    <requireSoapHeader name="To"
namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
    <requireActionHeader />
 </policy>
...
</policies>
```

In this policy file example, the **<Action>**, **<MessageID>**, and **<To>** elements are required on all incoming request messages. A custom policy assertion, **bodyValidator**, is specified in the **<extensions>** section (see the section, "Custom Policy Assertion — Message Body Validation," for sample code). You should indicate the namespace as appropriate for your project.

The **type** attribute for the **bodyValidator** extension declared in the preceding policy code example is formatted as the fully qualified class name (namespace + class name) followed by a comma and then the name of the assembly that contains the assertion class.

If you are not using policy to implement authentication and/or message protection for your service as previously described in this section, you must now enable the service to support WSE and enable policy support. WSE does not recognize custom policy assertions when it parses the policy cache file, and it will disable policy support if you attempt to configure it using the WSE Settings tool. If you have to enable policy support after you have added a custom policy assertion to your policy cache, you must add a **<policy>** element to the service's Web.config file to enable policy support.

```
<microsoft.web.services3>
...
   <policy fileName="wse3policyCache.config" />
...
</microsoft.web.services3>
```

Replace the value specified for the **fileName** attribute with the file path and name of your policy cache file.

Custom Policy Assertion — Message Body Validation

The following code example shows the custom policy assertion used to check the message body against an XML Schema (XSD).

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.IO;
using System.Xml.Schema;
using System.Web;
using System.Configuration;

using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Design;

namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.MessageValidation.CustomAssertions
{
    /// <summary>
    /// This Custom PolicyAssertion class validates the received SOAP body
    /// against an XML Schema (XSD) document whose path is configured in the
policy document.
    /// </summary>
    public class BodyValidatorAssertion : PolicyAssertion
    {
        private string xsdPath;
```

(continued)

(continued)

```
        public override SoapFilter CreateClientInputFilter(FilterCreationContext
context)
        {
            return null;
        }

        public override SoapFilter CreateClientOutputFilter(FilterCreationContext
context)
        {
            return null;
        }

        public override SoapFilter CreateServiceInputFilter(FilterCreationContext
context)
        {
            return new BodyValidatorAssertion.ServiceInputFilter(this);
        }

        public override SoapFilter CreateServiceOutputFilter(FilterCreationContext
context)
        {
            return null;
        }

        public override void ReadXml(System.Xml.XmlReader reader,
IDictionary<string, Type> extensions)
        {
            if (reader == null)
                throw new ArgumentNullException("reader");
            if (extensions == null)
                throw new ArgumentNullException("extensions");

            bool isEmpty = reader.IsEmptyElement;

            string xsdPath = reader.GetAttribute("xsdPath");
            if (!string.IsNullOrEmpty(xsdPath))
            {
                this.xsdPath = xsdPath;
            }
            else
            {
                throw new ConfigurationErrorsException(Messages.MissingXsdPath);
            }

            reader.ReadStartElement("bodyValidator");

            if(!isEmpty)
                reader.ReadEndElement();
        }
```

(continued)

(continued)

```
        public override void WriteXml(System.Xml.XmlWriter writer)
        {
            writer.WriteStartElement("bodyValidator");
            writer.WriteAttributeString("xsdPath", this.xsdPath);
            writer.WriteEndElement();
        }

        protected class ServiceInputFilter : SoapFilter
        {
            #region Custom Fields

            private XmlSchema schema;

            #endregion

            #region Constructors
            public ServiceInputFilter(BodyValidatorAssertion assertion)
            {
                string xsdPath = assertion.xsdPath;
                if (!Path.IsPathRooted(xsdPath))
                {
                   xsdPath =
Path.Combine(AppDomain.CurrentDomain.SetupInformation.ApplicationBase, xsdPath);
                }

                using (StreamReader streamReader = new StreamReader(xsdPath))
                {
                    this.schema = XmlSchema.Read(streamReader, ValidationHandler);
                    streamReader.Close();
                }
            }
            #endregion

            #region SoapFilter Methods
            public override SoapFilterResult ProcessMessage(SoapEnvelope envelope)
            {
                ValidationResults results = new ValidationResults();
                SoapContext.Current.MessageState.Set(results);

                ValidateSchema(envelope.Body.InnerXml);

                if (results.ErrorsCount > 0)
                {
                    throw new
ApplicationException(string.Format(Messages.ValidationError,
results.ErrorMessage));
                }

                return SoapFilterResult.Continue;
            }
            #endregion

            #region Custom Methods
            /// <summary>
            /// Performs the validation of the SOAP body against the specified XML
Schema (XSD) document.
            /// </summary>
            /// <param name="xmlDoc">SOAP message's body (XML)</param>
            public void ValidateSchema(string xmlDoc)
            {
                try
                {
                    XmlReaderSettings settings = new XmlReaderSettings();
                    settings.Schemas.Add(this.schema);
                    settings.ValidationType = ValidationType.Schema;

                    XmlReader reader = XmlReader.Create(new StringReader(xmlDoc),
settings);

                    // Validate the document.
                    while (reader.Read()) ;

                    reader.Close();
                }
                catch(Exception ex)
                {
                    throw new
ApplicationException(string.Format(Messages.SchemaValidationException,
ex.Message));
                }
            }

            /// <summary>
            /// Callback method that stores the error messages.
            /// </summary>
            /// <param name="sender"></param>
            /// <param name="args"></param>
            public void ValidationHandler(object sender, ValidationEventArgs args)
            {
                if (args.Severity == XmlSeverityType.Error)
                {
                    ValidationResults results =
SoapContext.Current.MessageState.Get<ValidationResults>();

                    results.ErrorMessage.Append(args.Message + "\r\n");
                    results.ErrorsCount++;
                }
            }
            #endregion

            private class ValidationResults
            {
                public StringBuilder ErrorMessage = new StringBuilder();
                public int ErrorsCount;
            }

        }

    }
}
```

In the preceding example, the **Messages.MissingXsdPath** refers to a resource string that provides a message for the **ConfigurationErrorsException** that is being thrown. As appropriate, you should substitute this and other resource strings used in the code example with a simple exception message to describe the nature of the exception.

Note: The validator assertion will only validate the structure of XML data in the message that has the same namespace as the schema that is used to validate it. Data with other namespaces is ignored for schema validation.

You should take care when using a policy assertion to validate an XML Schema (XSD) if a party other than the Web service developer will be responsible for configuring the service's policy when it is deployed into production. If the party responsible for configuring policy in production does not add the validation assertion, the validation will not be performed. If Web service development and policy configuration responsibilities are not held by the same individuals, you should consider using a helper class that is called from within the service to perform the validation instead. Alternatively, you can add the schema to the resource file for your project. In this case, the schema does not have to be deployed as a separate file. For more information, see Resolving the Unknown: Building Custom XmlResolvers in the .NET Framework on MSDN.

The policy assertion caches the schema in memory that it uses to validate incoming request messages. If you make changes to the schema, you may have to restart Microsoft Internet Information Services (IIS) to ensure that the updated schema is loaded into memory.

Use Regular Expressions to Parse Input

The following code example shows how to use regular expressions to parse input on the Web service to ensure that only valid characters are used. Place this code where it can be called to validate input, after the message has been decrypted (if message layer security is implemented). For example, the following code can be added to the service to validate each string input parameter.

```
...
using System.Text.RegularExpressions;
...
private bool Validate(string searchString)
{
Regex r = new Regex("^[0-9A-Za-z]{1,10}$");
       return r.IsMatch(searchString);
}
```

The preceding example provides a simple example for regular expression validation that does not allow any non-alphanumeric characters. Consequently, it may not be suitable for use in all applications. You can use more sophisticated checks for complex data, such as social security numbers and telephone numbers. For more information about implementing regular expressions, see How To: Use Regular Expressions to Constrain Input in ASP.NET on MSDN.

Note: Although the custom policy assertion provided in this pattern is applied after the security filters in the pipeline (that is, after the message has been decrypted), the regular expression code is not used in the policy assertion because it would require the policy assertion to have explicit knowledge of input parameters contained in the data.

You can also use regular expressions to validate user input on client applications. The main benefit of validating input from the client's perspective is to save a round trip to the Web service if data validation fails. For this approach to be effective, you must be able to validate data according to the Web service's validation requirements.

However, the service should never depend on the client to perform validation checks. You must always perform validation checks on the server, because an attacker could use a different client that does not perform the check or messages could be altered after a check has been performed at the client.

The following example demonstrates how regular expression validation can be described within an XML Schema (XSD). Regular expression validation that uses an XML Schema (XSD) allows the Web service publisher to indicate to consumers what the Web service expects. However, it does not perform as well as regular expression validation in code.

```
...
<xsd:simpleType name="CustomerReferenceType">
  <xsd:restriction base="xsd:normalizedString">
    <xsd:maxLength value="20"/>
<xsd:pattern value="[A-D][0-9]{5}-[0-9A-Z]{7}-[a-z]{3}#*"/>
</xsd:restriction>
</xsd:simpleType>
...
```

For more information about using regular expressions in XSD schemas, see XML Schema Regular Expressions on MSDN.

Parameterize SQL Queries

Web services often use a database to store and retrieve data. Web service request messages could contain malicious input to inject SQL commands into database queries. The following example provides an example of how to parameterize SQL queries. Whenever possible, you should use stored procedures for both performance and security reasons. Stored procedures accept input through parameters, and they generally work best to enforce minimum privilege for data retrieval and modification. The example shows how to parameterize dynamic SQL if your application must use it.

Note: The example assumes that a regular expression has already been used to validate the **searchString** parameter. For more information, see the previous section, "Use Regular Expressions to Parse Input."

```
...
using System.Data.SqlClient;
using System.Configuration;
...
private Customer[] GetCustomerList(string country, string searchString)
{
CustomerCollection customerCollection = new CustomerCollection();
Customer customer = new Customer();
using (SqlConnection conn = new
SqlConnection(ConfigurationManager.ConnectionStrings["Northwind"].ToString()))
{
string selectString = "SELECT * FROM Customers WHERE Country = @Country AND
(CompanyName LIKE '%' + @SearchString + '%' OR ContactName LIKE '%' +
@SearchString + '%' OR @SearchString IS NULL)";
    conn.Open();
    SqlCommand cmd = new SqlCommand(selectString, conn);
    cmd.Parameters.Add("@Country", SqlDbType.VarChar).Value = country;
    cmd.Parameters.Add("@SearchString", SqlDbType.VarChar, 10).Value =
searchString;
    SqlDataReader reader = cmd.ExecuteReader();
    while (reader.Read())
    {
     customer.CustomerID = reader["CustomerID"].ToString();
     customer.CompanyName = reader["CompanyName"].ToString();
     customer.ContactName = reader["ContactName"].ToString();
     customer.ContactTitle = reader["ContactTitle"].ToString();
     customer.Address = reader["Address"].ToString();
     customer.City = reader["City"].ToString();
     customer.Region = reader["Region"].ToString();
     customer.PostalCode = reader["PostalCode"].ToString();
     customer.Country = reader["Country"].ToString();
     customer.Phone = reader["Phone"].ToString();
     customer.Fax = reader["Fax"].ToString();
     customerCollection.Add(customer);
    }
    reader.Close();
    conn.Close();
   }
   return (Customer[])customerCollection.ToArray(typeof(Customer));
  }
```

In this example, the **Customer** and **CustomerCollection** classes are custom data objects. As appropriate, replace the data objects and SQL query for your application. The important point is to parameterize the query instead of directly concatenating input into the SQL query.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The majority of attacks that result from malformed messages, invalid characters, or SQL injection are mitigated with the approach outlined in this implementation pattern.

Liabilities

The liabilities associated with the Implementing Message Validation in WSE 3.0 pattern include the following:

- Validating messages against very large schemas can affect system performance. Typically, the cost of parsing is multiplied two to four times when the schema validation is performed on an XML message. For more information about XML performance guidance in the .NET Framework, see Chapter 9, Improving XML Performance in *Improving .NET Application Performance and Scalability* on MSDN.

  If message schema validation is causing performance problems, you should consider the following optimizations:

  - Make sure that you are reading your schemas only once from the schema file, and cache them in memory to minimize I/O.
  - Reduce the message schema to essential elements that are required for a particular Web service or Web service operation. Another option is to use regular expression validation in code to validate structural elements.
  - Incorporate more sophisticated regular expression checking. The regular expression validation example provided in this application is very strict and does not account for validation requirements specific to your service. A thorough threat analysis of your application should reveal any need for a specific form of regular expression checking. For more information about validating input with regular expressions, see How To: Use Regular Expressions to Constrain Input in ASP.NET on MSDN.

Security Considerations

Security considerations associated with the Implementing Message Validation in WSE 3.0 pattern include the following:

- Attackers may attempt to work around message validation. You should be aware of known attempts to work around message validation and adjust your validation code accordingly. Keep your platform up to date with the latest security updates to mitigate issues with built-in security features.
- Schema validation validates only basic data types, such as integers, dates, and structures; it should always be supplemented with regular expression validation. You can directly implement regular expression validation in the XML Schema (XSD) or in code to validate more complex data, such as social security numbers and telephone numbers. Regular expression validation directly in the XML Schema (XSD) is useful to communicate what the service requires as valid input to client applications, but it does not perform as well as regular expressions implemented in code.

Exception Shielding

Context

A client is accessing a Web service. The Web service is designed according to the principals of service orientation, which ensures that the boundaries of the service are explicit, and requires that exception information related to the internal implementation of the service is managed within the service.

Problem

How do you prevent a Web service from disclosing information about the internal implementation of the service when an exception occurs?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Exception details may contain clues that an attacker can use to exploit resources used by the system**. Detailed fault messages can disclose information about the Web service or resources accessed by the Web service code that threw the exception. An attacker may deliberately cause the Web service to throw an unhandled exception in an attempt to obtain sensitive information, such as connection strings, server names, SQL queries, XPath commands, stack traces, and data schemas. The attacker can then use this information to exploit the Web service or the resources that it accesses.
- **Information related to anticipated exceptions needs to be returned to the client.** In cases where an exception is expected, an error message that does not contain sensitive internal information can be returned to the client. A service may provide information about the cause of the fault, where the information is not considered a security risk. In some cases (for example, data validation errors), the potential savings in administrative support may outweigh the risk of providing the requestor with more detailed information about an exception.

The following condition is not resolved by the base pattern, but it is resolved by Extension 1 — Logging Exceptions:

- **Exceptions that occur within a Web service should be logged to support troubleshooting**. Information within an exception can be used by monitoring tools to automatically notify system administrators when an exception occurs. The same information can also be used by application developers to diagnose exceptions that occur within the logic of the service or with resources that the service is dependent on. In some cases, you may require that an error message that is returned to the client contains an ID that helpdesk staff can use to troubleshoot user problems.

For more information, see the “Extensions” section at the end of this pattern.

Solution

Use the Exception Shielding pattern to sanitize unsafe exceptions by replacing them with exceptions that are safe by design. Return only those exceptions to the client that have been sanitized or exceptions that are safe by design. Exceptions that are safe by design do not contain sensitive information in the exception message, and they do not contain a detailed stack trace, either of which might reveal sensitive information about the Web service's inner workings.

Participants

Exception shielding involves the following participants:

- **Client**. The client application that calls a Web service.
- **Service**. The Web service that processes requests that are received from clients.

Process

Figure 5.6 illustrates how an unhandled exception that is thrown by a Web service is processed by a service that implements exception shielding.

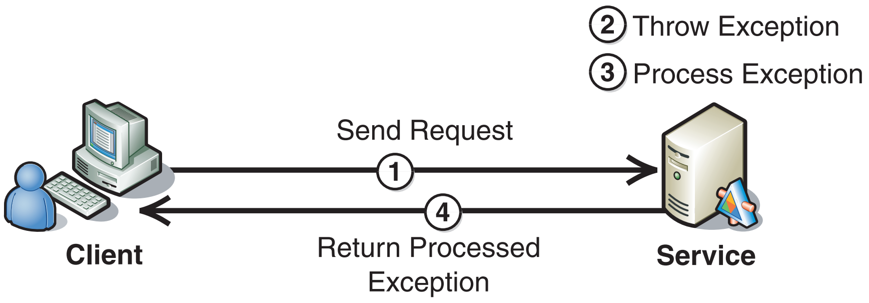


Figure 5.6
A Web service that implements exception shielding

As illustrated in Figure 5.6, the exception shielding process is described in the following steps:

1. **The client submits a request to the service**.
2. **The service attempts to process the request and throws an exception**. The exception can be safe or unsafe by design.
3. **Exception shielding logic processes the exception**. If the exception type is safe by design, it is already considered sanitized and is returned to the client unmodified. If the exception is unsafe, the exception is replaced with an exception that is safe by design, which is returned to the client.
4. **The service returns the processed exception to the client**. The exception is wrapped in a SOAP fault before it is returned to the client.

Example

Global Bank has designed a Web service that checks the balance of customer accounts. Global Bank needs to ensure that when exceptions occur, information potentially useful to attackers is not revealed.

For some anticipated exceptions that are safe by design, such as data validation errors, the Web service returns appropriate information to the client. For other exceptions, such as authentication failures, the exception logic sanitizes the exception, replacing it with an exception that is safe by design.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Exception Shielding pattern include the following:

- Exception shielding prevents sensitive information from being disclosed in exception details.
- Maintenance staff can enable detailed exception information to be returned by production Web services. This allows them to troubleshoot issues in the production environment without exposing exception details to external consumers.
- Unanticipated exceptions that are thrown by Web services in the enterprise can be uniformly and centrally managed. Different Web services that implement disparate methods of exception management make it more difficult for enterprise architects to ensure that unhandled exceptions are managed securely and consistently across an enterprise.

Liabilities

Adding exception shielding logic to a Web service increases the amount of processing the service must perform. You must ensure that exception shielding is performed efficiently. Any related activities, such as logging, may need to be minimized to prevent the service from becoming a performance bottleneck.

Security Considerations

Security considerations associated with the Exception Shielding pattern include the following:

- Unhandled exceptions may be wrapped by another exception. You should ensure that the outer exception and all wrapped exceptions are checked by the exception shield logic before they are returned to a Web service client.
- You should use exception handling throughout the entire application's code base. This prevents internal implementation details of the service from being revealed to the client.
- The "deny" model is an alternative to the "allow" model that is used in the Exception Shielding pattern. In the deny model, specific exceptions are registered to be sanitized, and all other exceptions are sent back to the client unmodified. However, the deny model is considered less secure, because unanticipated exceptions are not sanitized.

Extensions

The extension described here builds on the base pattern to provide additional capabilities. In addition to resolving the forces stated for the base pattern, this extension also resolves the following condition:

- **Exceptions that occur within a Web service should be logged to support troubleshooting**. Information in an exception can be used by monitoring tools to automatically notify system administrators when an exception occurs. The same information can also be used by application developers to diagnose exceptions that occur within the logic of the service or with resources that the service is dependent on. In some cases, you may require that an error message that is returned to the client contains an ID that helpdesk staff can use to troubleshoot user problems.

Extension 1 — Logging Exceptions

In addition to processing exceptions, the exception shielding logic can also log the full details of the exception to an event log. This allows maintenance staff to identify and troubleshoot the exceptions. The information also assists with intrusion detection and incident response.

The exception shielding logic can also generate an exception identifier for each exception and pass it back to the client in a message, so that it can be presented to the user in the form of an error message. This allows the exception that is returned to the client to be directly traced to detailed exception information located in the event log, which can assist in dealing with helpdesk calls.

Related Patterns

The following child pattern is related to the Exception Shielding pattern:

- **Implementing Exception Shielding**. This pattern provides implementation steps and recommendations for using exception shielding.

Implementing Exception Shielding

Context

You are implementing a Web service that runs on the .NET Framework. You must ensure that exceptions thrown by the Web service do not disclose sensitive information about the service or resources that it accesses.

Objectives

The objectives of this pattern are to:

- Prevent the Web service from disclosing sensitive information in exception messages.
- Create exceptions that are safe by design in which exception information is returned to Web service clients.
- Write unsanitized exception details to a log to support monitoring and troubleshooting the Web service.

Content

This pattern consists of the following sections:

- **Implementation Strategy**. This section provides a high-level description of the strategy used to implement the solution that includes descriptions of the participants and the process.
- **Implementation Approach**. This section describes the following steps that are required to implement the Exception Shielding pattern:
  - Create a custom exception class.
  - Enclose code in **try/catch** blocks.
  - Create a method that sanitizes exceptions.
- **Resulting Context**. This section outlines the benefits, liabilities, and security considerations when the pattern is implemented.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

The strategy for the implementation of this pattern includes the following:

- Implement a custom exception to return sanitized exception data to the client that does not reveal sensitive information about the Web service, such as database connection strings and resource URLs.
- Enclose all code in **try/catch** blocks. Handle the custom "safe" exceptions first, such as business exceptions derived from a custom "safe" exception type, and then handle all other exception types and run them through the sanitization process. After an exception is sanitized, it proceeds up the stack back to the client.

Note: To fully understand this pattern, you must have some familiarity and experience with the .NET Framework.

Participants

The Exception Shielding pattern involves the following participants:

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to authorizing the client.

Process

The Exception Shielding pattern describes the process to prevent detailed exception information from returning to a client. This implementation pattern provides a detailed description of that process that is specific to the implementation.

Figure 5.7 illustrates how a Web service processes messages exception details from an exception.

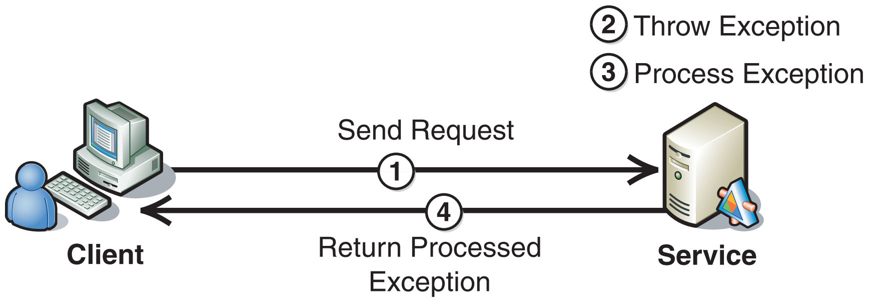


Figure 5.7
A Web service throwing and processing an exception.

The process uses the following steps:

1. **The client submits a request to the service**.
2. **The service attempts to process the request and throws an exception**. The exception could be safe by design or unsafe.
3. **Exception shielding logic processes the exception**. If the exception type is safe by design, it is considered sanitized and the service can return it to the client unmodified. If the exception is unsafe, it is replaced with an exception that is safe by design, which the service can return to the client.
4. **The service returns the processed exception to the client**. The sanitized exception that the service returns is wrapped in a SOAP Fault. The following Web Services Enhancements (WSE) message trace provides an example of what a sanitized exception returned by the service would look like on the wire in a response to the client.

```
<soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>System.Web.Services.Protocols.SoapException: Server was
unable to process request. ---&gt;
Microsoft.Practices.WSSP.WSE3.QuickStart.ExceptionShielding.CustomExceptions.Cl
ientException: An error has occurred while consuming Web service. Please
contact your administrator for more information. ErrorId: acba7202-ef5f-4921-
bbfa-b7a787e3ad53
  at
Microsoft.Practices.WSSP.WSE3.QuickStart.ExceptionShielding.Service.Service.Hel
loWorld()
  --- End of inner exception stack trace ---</faultstring>
      <detail />
     </soap:Fault>
```

The exception information includes the fully qualified name of the sanitized exception class, a sanitized exception message, and the location in the stack where the sanitized exception was thrown. The exception class and limited stack information in the SOAP Fault do not translate to a physical location on the service. However, if you have determined after a thorough threat analysis of the service application that these two items of data may contain sensitive information, you may have to take further steps to sanitize the exception. For more information about this topic, see the "Security Considerations" section.

Implementation Approach

This section describes how to implement this pattern. Exception shielding occurs on the service, which is the focus of the implementation. The following steps describe the tasks necessary to implement exception shielding on the service:

1. Create a custom exception class.
2. Enclose code in **try/catch** blocks.
3. Create a method that sanitizes exceptions.

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

Create a Custom Exception Class

Derive a custom exception class from **Exception** to create an exception type that is defined as safe by design. The following code example provides a custom exception class named **ClientException**.

```
using System;

namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.ExceptionShielding.CustomExceptions
{
    public class ClientException : Exception
    {
        public ClientException(string message) : base(message)
        {
        }
    }
}
```

In this code sample, the **ClientException** class receives a generic exception message that is defined in the service's Web.config file as an argument in its constructor. The generic exception message is intended to notify the client that an error has occurred without providing details of the exception stack or the exception.

Enclose Code in Try/Catch Blocks

The following code sample provides an example of how exceptions are handled based on whether they are considered safe or unsafe. In this example, the only exception type that is considered safe by design is the **ClientException** class. The application may throw other exceptions derived from this class that are returned to the client in an unsanitized state. All other exceptions are sanitized by converting them into a **ClientException**.

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Configuration;
using System.Diagnostics;

using CustomExceptions;

using Microsoft.Web.Services3;
```

(continued)

(continued)

```
namespace ExceptionShielding.Service
{
 [WebService(Namespace = "http://tempuri.org/")]
 [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
 [Policy("ServicePolicy")]
 public class Service : System.Web.Services.WebService
 {
  ...

  [WebMethod]
  public string HelloWorld()
  {
   try
   {
    // Executes an operation, which can return an exception
    DoSomething();
   }
   catch(ClientException)
   {
    //This exception is safe, so it is returned without any change
    throw;
   }
   catch(Exception unsafeException)
   {
    ClientException clientException = GetSanitizedException(unsafeException);
    throw clientException;
   }

   return "Hello World";
  }

  /// <summary>
  /// Executes a simple operation
  /// </summary>
  private void DoSomething()
  {
   Random rnd = new Random();
   if (rnd.Next(1, 10) > 1)
   {
    throw new
System.Security.SecurityException(ConfigurationManager.AppSettings["SystemExceptio
nMessage"]);
   }
   else
   {
    throw new
ClientException(ConfigurationManager.AppSettings["ClientExceptionMessage"]);
   }
  }
}
```

In the preceding example, the Web service does something to cause an error. In the **DoSomething()** method, different error types are randomly thrown to demonstrate how exceptions that are safe by design are handled differently from those that have to be sanitized. If a **ClientException** is thrown, it is returned to the client unsanitized. Any other exception types that are not of this class or derived from it are sanitized by the **GetSanitizedException** method described in the following section.

Note: Exceptions should be sanitized as far up the Web service call stack as possible to minimize the stack information that is returned in the sanitized exception. If possible, sanitized exceptions should be thrown from the Web method processing the request from the client.

Create a Method that Sanitizes Exceptions

If any type of exception thrown is not a **ClientException** or derived from this class, the **GetSanitizedException()** method is called to return a sanitized **ClientException**. The **ClientException** is then returned to the client. The details of the unsanitized exception are captured in the application log for troubleshooting. The following code example provides an example of sanitizing unsafe exceptions.

```
/// <summary>
/// Logs the original exception and returns a more generic exception with a
reference number.
/// </summary>
/// <param name="exception"></param>
/// <returns></returns>

[EventLogPermissionAttribute(System.Security.Permissions.SecurityAction.Demand,
PermissionAccess=EventLogPermissionAccess.Administer)]
private ClientException GetSanitizedException(Exception exception)
{
   string errorId = Guid.NewGuid().ToString();
   string errorMessage = string.Format(Resources.Messages.ExceptionThrownMessage,
             errorId, exception.Message);

   string source = ConfigurationManager.AppSettings["ApplicationName"];
   if (string.IsNullOrEmpty(source))
   {
      source = AppDomain.CurrentDomain.FriendlyName;
   }


   try
   {
      // Logs the original exception.
      EventLog.WriteEntry(
                  source,
                  errorMessage,
                  EventLogEntryType.Error);
   }
```

(continued)

(continued)

```
   catch
   {
      // Uses an alternative event log source in case of error.
      string alternativeSource = String.Format("ASP.NET {0}.0",
Environment.Version.ToString(3));

      // Swallowing exceptions like this is generally considered a bad practice -
but the alternative is to
      // possibly return exception information unshielded. Users should consider
the benefits
      // and liabilities in their own environments.
      EventLog.WriteEntry(alternativeSource,
                    String.Format(Resources.Messages.InsufficientPermissions,
source),
                    EventLogEntryType.Warning);

      EventLog.WriteEntry(alternativeSource,
                    errorMessage,
                    EventLogEntryType.Error);
   }

   // Returns an exception with a generic message.
   return new
ClientException(String.Format(ConfigurationManager.AppSettings["ClientExceptionGen
ericMessage"], errorId));
}
```

The code example that sanitizes exceptions uses the following configuration settings, which are defined in the application's configuration file.

```
...
<appSettings>
   <add key="ApplicationName" value="ExceptionShieldingService"/>
   <add key="ClientExceptionGenericMessage" value="An error has occurred while
consuming Web service. Please contact your administrator for more information.
ErrorId: {0}"/>
    <!-- Example text for exceptions that were not safe by design. Exception
shielding should sanitize the login and password from the exception information --
>
    <add key="SystemExceptionMessage" value="SqlError:An exception has occurred.
Cannot connect to database using login='Bob' and password='password'"/>
    <!-- Example text for exceptions that are safe by design -->
    <add key="ClientExceptionMessage" value="This exception is inoffensive and is
not being sanitized."/>
</appSettings>
...
```

This implementation uses an **EventLog** instance to log exception details. Depending on your requirements, you may have to use a different logging mechanism. If you use a different one, replace the logging code in the **GetSantizedException()** method with the appropriate code for your log implementation. By default, this example writes to the application event log.

The preceding code example uses an **EventLogPermissionAttribute** to ensure that the code has the ability to write to the event log. If the assembly in which the sample code is running does not have this permission based on code access security settings, the assembly will not load at run time.

The ASP.NET account does not have the required permissions to create a new source for the event log. The preceding code example specifies itself as the source of the unsanitized exception details that are written to the event log. If you are not running your Web service under a custom service account with permissions to create sources for the event log, you have two options to resolve this issue:

- Grant permissions to the default service account to create new event sources. Usually, you should not use this approach because it gives any application running under the default service account the ability to create event sources at any time.
- Create the event source that the Web service uses beforehand. For example, you can use an installer application running under an account with the appropriate security permissions. For more information about this topic, see "Creating a New Event Source at Install Time" in How To: Use the Network Service Account to Access Resources in ASP.NET.

If an event source has not been registered for the sample code, it will fail when it attempts to write the exception to the event log. In this case, it will attempt to write to the default ASP.NET run-time event source as a fallback measure, so that some record of the exception can be captured for troubleshooting.

Functionality has been added for a Web service publisher that provides support to Web service consumers to assist customers in identifying an error for troubleshooting. The particular occurrence of the exception that is thrown is assigned a unique identifier. The unique identifier is returned to the client in the sanitized exception to assist Web service support staff in finding the unsanitized exception details in the log to diagnose the error.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

Only exceptions that are considered safe by design are returned to the calling application. This allows finer control over the exposure of the Web service's internal information.

Liabilities

Although this implementation handles a scenario in which an event source for the service is not registered for the event log, it does not address the scenario where other types of exceptions are thrown while sanitizing unsafe exception types.

Security Considerations

Security considerations associated with the Implementing Exception Shielding pattern include the following:

- If an attacker finds a way to intentionally cause exceptions, the attacker may use it to attempt a denial of service attack or flood the application log with bogus exceptions. You can reduce this threat by limiting more resource-intensive processing of sanitized exception types. For example, logging causes I/O operations that may impact the performance of the application while it is sanitizing exceptions.

  You should consider logging information only on certain types of sanitized exceptions that are most essential to log for security or troubleshooting purposes. However, carefully balance this approach to mitigating the problem with your auditing and logging requirements.
- This implementation can only sanitize exceptions thrown within the Web service implementation. It does not sanitize exceptions thrown higher up in the application stack or in the communication pipeline. WSE 3.0 limits this behavior by minimizing the information returned in exceptions thrown from within the WSE 3.0 pipeline through the **<detailedErrors>** configuration setting. By default, the **Enabled** attribute of the **<detailedErrors>** setting is **False**, so you do not have to explicitly enable it.
- The sanitized exceptions thrown by an implementation of this pattern eliminate potentially sensitive information in unsanitized exceptions, but the sanitized exceptions themselves do contain the fully qualified class name of the sanitized exception (for example, **Microsoft.Practices.WSSP.WSE3.QuickStart.ExceptionShielding .CustomExceptions.ClientException**), and the exact point in the stack trace where the exception was sanitized, (for example, **Microsoft.Practices.WSSP.WSE3.QuickStart.ExceptionShielding.Service .Service.HelloWorld()**).

Usually, this information is considered harmless, but it may be considered sensitive in certain circumstances. If a thorough threat analysis determines that this information is sensitive, remove it by building and deploying a custom WSE 3.0 policy assertion to remove the information from the SOAP Fault before it is returned to the client. For more information about creating custom policy assertions in WSE 3.0, see Custom Policy Assertions on MSDN.

- If you must return to the client sensitive information contained in exceptions, there are additional options available to protect the exceptions. If you implement message layer security, you can protect fault messages by setting the **encryptBody** attribute of the **<fault>** element to **true** in the turnkey assertion configuration. If you are providing message protection at the transport layer, communications between the client and service are encrypted anyway.
- If you enable debugging for the service through ASP.NET configuration, it may reveal additional information through sanitized exceptions about the location in code where the exception occurred. This information may be considered sensitive in nature because it provides information about the physical location of the code on the server where the exception occurred. If you do not want this type of information revealed to clients through sanitized exceptions, make sure that ASP.NET debugging is disabled on the service.

More Information

For more information about idempotent methods, see “9 Method Definitions”: *http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*.

For more information about idempotent, see “Idempotent” on the Wikipedia Web site: *http://en.wikipedia.org/wiki/Idempotent*.

For more information about idempotent Web services, see “Idempotent Receiver“ on the Enterprise Integration Patterns Web site: *http://www.eaipatterns.com /IdempotentReceiver.html*.

For more information about SOAP Message Security, see OASIS: “Web Services Security: SOAP Message Security 1.0 (WS Security 2004)”: *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.

For more information about SQL Server performance optimization, see “Optimizing Database Performance Overview” on MSDN: *http://msdn.microsoft.com/library/?url= /library/en-us/optimsql/odp_tunovw_9mxz.asp?frame=true*.

For more information about security best practices for SQL Server 2000, see “SQL Server 2000 SP3 Security Features and Best Practices” on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/sp3sec00.mspx*.

Chapter 4, “Design Guidelines for Secure Web Applications,” in *Improving Web Application Security: Threats and Countermeasures* on MSDN: *http://msdn.microsoft.com /library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh04.asp*.

For more information about **<httpRuntime>**, see "**<httpRuntime>** Element" in the *.NET Framework General Reference* on MSDN: *http://msdn.microsoft.com/library /default.asp?url=/library/en-us/cpgenref/html/gngrfhttpruntimesection.asp*.

For more information about WSE 3.0 policy assertions, see "Policy Assertions" on MSDN: *http://msdn.microsoft.com/library/?url=/library/en-us/wse3.0/html /1d3257fd-fcfb-45cf-beca-3cfcefceaa8b.asp*.

For more information about using the **SoapClient/SoapService** classes for messaging, see "How To: Send and Receive a SOAP Message by Using the **SoapClient** and **SoapService** Classes," in the WSE 3.0 documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html /8cbdb522-0672-4c17-b68e-0d3e65067271.asp*.

For more information about adding a schema to a resource file see "Resolving the Unknown: Building Custom XmlResolvers in the .NET Framework," on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnxmlnet/html /CusXmlRes.asp*.

For more information about implementing regular expressions, see "How To: Use Regular Expressions to Constrain Input in ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html /PAGHT000001.asp*.

For more information about using regular expressions in XML Schemas, see "XML Schema Regular Expressions" on MSDN: *http://msdn.microsoft.com/library /default.asp?url=/library/en-us/xmlsdk/html/ea72d044-6b46-4124-b6dc-95976e411b4a.asp*.

For more information about XML performance guidance in the .NET Framework, see Chapter 9, "Improving XML Performance," in *Improving .NET Application Performance and Scalability* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us /dnpag/html/scalenetchapt09.asp*.

For more information about how to create the event source that the Web service uses, see the "Creating a New Event Source at Install Time" section of "How To: Use the Network Service Account to Access Resources in ASP.NET" on Microsoft MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html /PAGHT000015.asp*.

For more information about creating custom Policy Assertions in WSE 3.0, see "Custom Policy Assertions" in the WSE 3.0 product documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html /5636c932-30d0-42c6-ac17-88c40b5935b8.asp*.

6

Service Deployment Patterns

Introduction

One or more Web services are most easily deployed on an application server, such as Windows Server 2003, that hosts the Web service. Frequently, the application server then communicates with other resources, such as database servers, and in some cases, other application servers that contain data for the Web service to process.

As organizations consider externally exposing Web services, there is often a reluctance to deploy the application server hosting the Web service in the perimeter network that external applications can access. However, Web service standards are designed for this scenario through the use of message layer security and intermediaries that can inspect message content and perform message validation and routing capabilities. Intermediaries can be used to supplement existing firewall devices, which are often used to protect an organization's perimeter network.

This chapter includes a design pattern for a perimeter service router, which acts as an intermediary that can be deployed in your perimeter network and route messages to a Web service endpoint that resides on an internal network that is invisible to the client. It also includes an implementation pattern that shows how the perimeter service router can be implemented using Microsoft technologies. The implementation pattern also contains variations from the core design pattern that show how the intermediary can perform actions such as message validation in addition to routing. These patterns are the following:

- Perimeter Service Router
- Implementing Perimeter Service Router in WSE 3.0

Perimeter Service Router

Context

External applications require access to one or more Web services that are deployed within a private network. Access to the Web services and resources in the private network is restricted to authenticated users. External applications should not have access to resources used by the Web services in the private network.

Problem

How do you make Web services in a private network available to external applications without exposing resources in the private network?

Forces

Any of the following conditions justifies using the solution described in this pattern:

- **Internal Web services and dependent resources may be targeted by attackers who are external to the network**. The organization must protect Web services on the internal network, so that any attacks do not affect the internal Web services or dependent resources.
- **Attackers can gain information about the internal network, and use it to compromise the network**. The organization must not reveal information about the internal network infrastructure that can be useful to attackers.

The following condition is an additional reason to use the solution:

- **External clients need reliable access to fixed service endpoints**. The location of a Web service's internal implementation may need to change dynamically to cater for the availability of dependent resources, or to cater for maintenance and batch processing windows. External clients should be unaffected by these changes.

Solution

Design a Web service intermediary that acts as a perimeter service router. The perimeter service router provides an external interface on the perimeter network for internal Web services. It accepts messages from external applications and routes them to the appropriate Web service on the private network.

Participants

Using the Perimeter Service Router pattern involves the following participants:

- **External application**. An application located outside of the private network that needs to access the Web services in a private network.
- **Perimeter service router**. The perimeter service router is a Web service that provides access to Web services in the private network.
- **Service**. One or more Web services that are accessed by the perimeter service router.

Figure 6.1 shows a perimeter service router accepting requests from a client and routing them to other services.

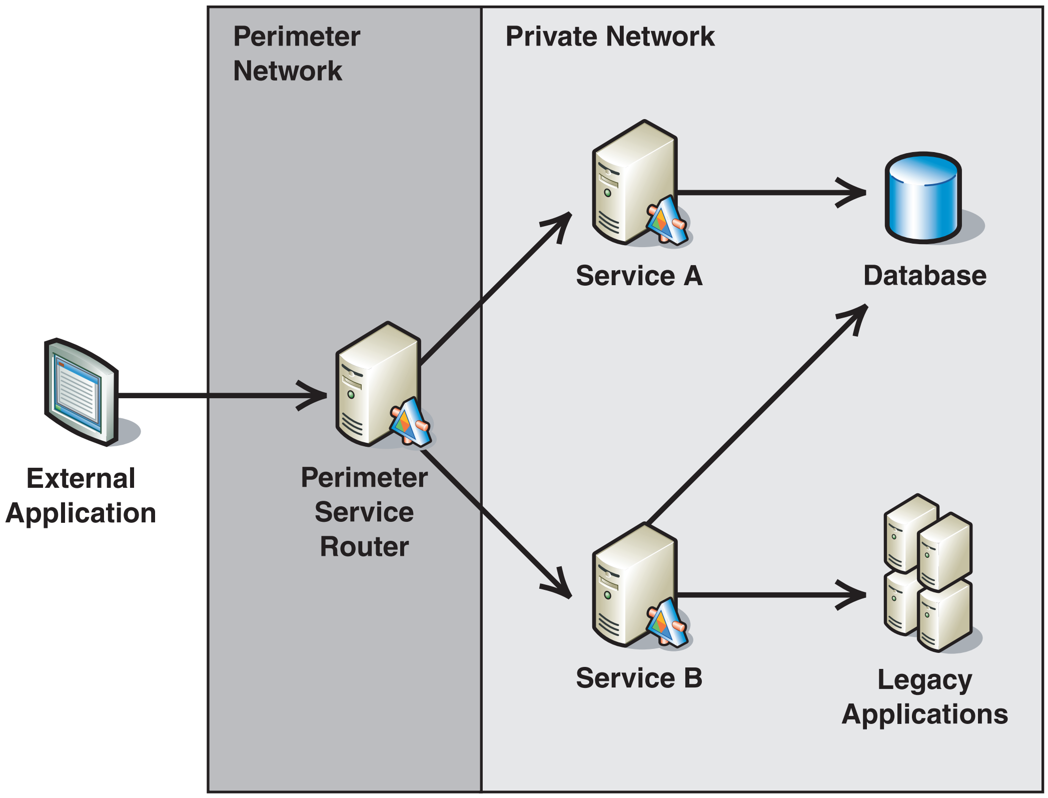


Figure 6.1
A perimeter service router on the perimeter network

The perimeter service router provides an entry point that external applications use to access the functionality exposed by internal services. The perimeter service router is typically deployed in a perimeter network (also known as DMZ or demilitarized zone), which has access to resources in the private network through a firewall. A perimeter service router operates at the application layer, and is intended to work in conjunction with existing firewall technologies and not to replace them.

Process

The following diagram illustrates the functionality of the perimeter service router.

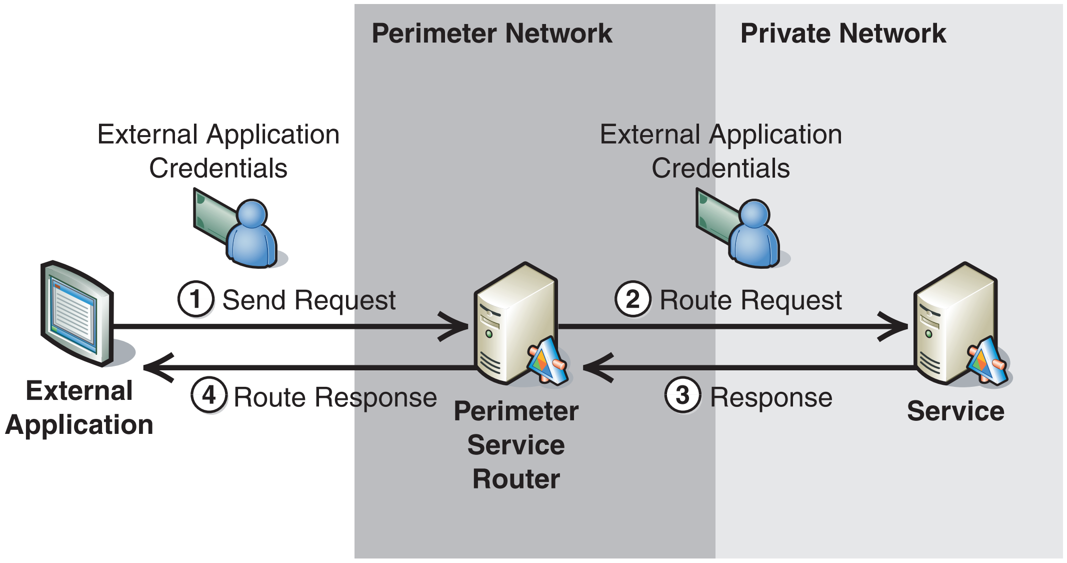


Figure 6.2

The functionality of the perimeter service router

As illustrated in Figure 6.2, the functionality of the perimeter service router is described in the following steps:

1. **The external application sends a request message**. The request message is addressed to the service's external interface on the perimeter service router. The perimeter service router typically "hides" the internal endpoint address by accepting requests through an external endpoint address that is exposed to external applications.
2. **The perimeter service router forwards the request message to the service**. The message is forwarded to the appropriate endpoint address. If the perimeter service router provides an external interface for multiple services on the private network, it will route the request to the appropriate service request based on the specific address where the request was sent.
3. **The service sends a response**. The service performs any security checks, such as authentication, and then processes the request. Based on the contract between the external application and the service, the service may send a response back to the external application.
4. **The perimeter service router forwards the response to the external application**. If the server sends a response in Step 3, the perimeter service router forwards the response to the external application.

Note: The basic perimeter service router described previously does not perform security functions as an intermediary such as authentication, replay detection or message validation. For more information about the security functions performed on a perimeter service router, see the "Benefits" section.

Example

Northwind Traders is a manufacturer that has created a suite of Web services that provide the ability to view and manage their inventory. Currently these services are only accessible to clients through a Web application provided by Northwind Traders. Many of Northwind's clients are retailers that also provide applications for their customers to order products online. When the retail customers order a Northwind Trader's product it is not possible to determine if that product is available prior to making the order. As a result, Northwind's clients would like direct access to the Web services that provide inventory information.

Instead of providing direct access to the inventory services, Northwind has decided to implement a perimeter service router that external clients can access. External clients can now incorporate calls to the perimeter service router directly into their applications to provide inventory information to their customers.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Perimeter Service Router pattern include the following:

- Security can be maintained at the perimeter service router, which provides an extra layer of security to protect the Web services.
- Servers that host internal Web services can be taken offline for maintenance without affecting the external interface. This can be accomplished by configuring the perimeter service router to start routing messages to a backup server while the maintenance is being performed.
- The perimeter service router represents a single point of entry for external clients. This allows it to be extended to support additional operations that external clients require. These requirements could include:
  - **Protocol Transition**. External clients can be authenticated with different mechanisms, such as X.509 certificates, or custom authentication that is validated against a database. After the external client has been authenticated, it can be transitioned into an internal protocol, such as the Kerberos version 5 protocol to access internal Web services.

- **Message Validation**. Request messages from external clients can be validated to make sure that they do not contain malicious content prior to sending them to an internal service. Message signatures can also be validated to detect tampering.
- **Exception Shielding**. Detailed error messages that are returned by internal services can be filtered or modified prior to sending responses back to external clients.
- **Replay Detection**. The perimeter service router can keep a cache of requests and reject any duplicate requests that are sent to the interface.
- **Message Transformation**. Request messages received from clients can be transformed into a structure that internal Web services require. This provides the ability to modify internal interfaces without affecting external interfaces. It is also possible to support several structures from external clients that can be mapped into an internal structure.
- **Auditing**. Activities may need to be attributed to a specific user or organization for accounting or security auditing purposes.

Note: In some cases, you need to provide some or all of these additional requirements for internal clients as well. In these cases, you need to place the logic that provides these functions on the internal network, or ensure that the internal clients also pass through the perimeter service router.

Liabilities

The liabilities associated with the Perimeter Service Router pattern include the following:

- Many platforms make exposing the application functionality simple. However, this can lead to a poor decision in terms of granularity. If the service interface is overly fine-grained, you can end up making too many calls to perform a specific action. You need to design your service interfaces to be appropriate for network or out-of-process communication.
- Each additional service interface that a service provides increases the amount of work required to make changes to the functionality that is exposed by a service.
- The Perimeter Service Router pattern adds complexity and performance overhead that may not be justified for very simple service-oriented applications.
- The perimeter service router may become a bottleneck when routing large numbers of messages. To avoid this problem, the perimeter service router should be designed with good performance as a high priority.

Security Considerations

Security considerations associated with the Perimeter Service Router pattern include the following:

- The perimeter service router is often the only point of entry to the internal network for external clients. This can make it a prime target for attackers. To guard against an attack, you must harden the platform on which the perimeter service router is deployed.
- Although the perimeter service router can provide an extra layer of security between external clients and internal Web services on a private network, you should still ensure that you design secure Web services on the internal network. You should also ensure that communications between the perimeter service router and internal Web services are secured.

Related Patterns

The following child pattern is related to the Perimeter Service Router pattern:

- Implementing Perimeter Service Router in WSE 3.0. This pattern provides steps and recommendations to implement a perimeter service router in WSE 3.0. It also discusses extensibility points in the **SoapHttpRouter** class in WSE 3.0 that you can use to address advanced scenarios, such as validation and dynamic routing

Implementing Perimeter Service Router in WSE 3.0

Context

You are exposing Web services deployed in a private network to external applications. Access to the Web services and resources in the private network is restricted to authenticated users. Any applications external to the private network must use a perimeter service router to access the Web services and resources deployed in the private network.

Objectives

The objectives of this pattern are to:

- Use a perimeter service router to provide an additional layer of security for services exposed to external clients.
- Allow the perimeter service router to route information to internal Web services based on a location contained within a configuration file.
- Demonstrate how to implement a perimeter service router using the WSE 3.0 **SoapHttpRouter** class.
- Discuss extensibility points in the **SoapHttpRouter** class in WSE 3.0 that you can use to address advanced scenarios, such as validation and dynamic routing.

Content

This pattern consists of the following sections:

- **Implementation Strategy**. This section provides a high-level description of the strategy to implement a perimeter service router.
- **Implementation Approach**. This section describes the steps required to implement this pattern:
  - General setup
  - Configure the external application
  - Configure the service router
  - Configure the service
- **Resulting Context**. This section outlines the benefits, liabilities, and the security considerations related to this pattern.
- **Extensions**. This section describes how to extend the base pattern to add more functionality for the router, including security policy enforcement.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

The implementation strategy for this pattern includes the following:

- Implement a perimeter service router in WSE 3.0, and deploy it as the service boundary between the perimeter network and the private network.
- Configure a handler to forward incoming requests from the router to the service.
- Create a routing referral cache that specifies the endpoint Uniform Resource Identifier (URI) of the service.

As an intermediary, the presence of the perimeter service router is not known to external applications. The service router represents the outward interface for the service that is deployed in the private network. External applications appear to communicate directly with the target service deployed to the internal network, although in reality they use an external URI for the service that is provided by the perimeter service router.

The Perimeter Service Router design pattern describes the perimeter service router as an intermediary that decouples internal services from external applications. This implementation pattern provides a more detailed description of that process.

Note: To fully understand this pattern, you must have some familiarity and experience with the .NET Framework, WSE 3.0, and Web service development.

Participants

Implementing a perimeter service router in WSE 3.0 involves the following participants:

- **External application**. An application outside the private network that needs to access the Web services in the private network.
- **Perimeter service router**. A Web service that provides access to Web services in the private network.
- **Service**. One or more Web services that the perimeter service router can access.

The following diagram displays the participants and their relation to each another in the private network, the perimeter network, and the public network.

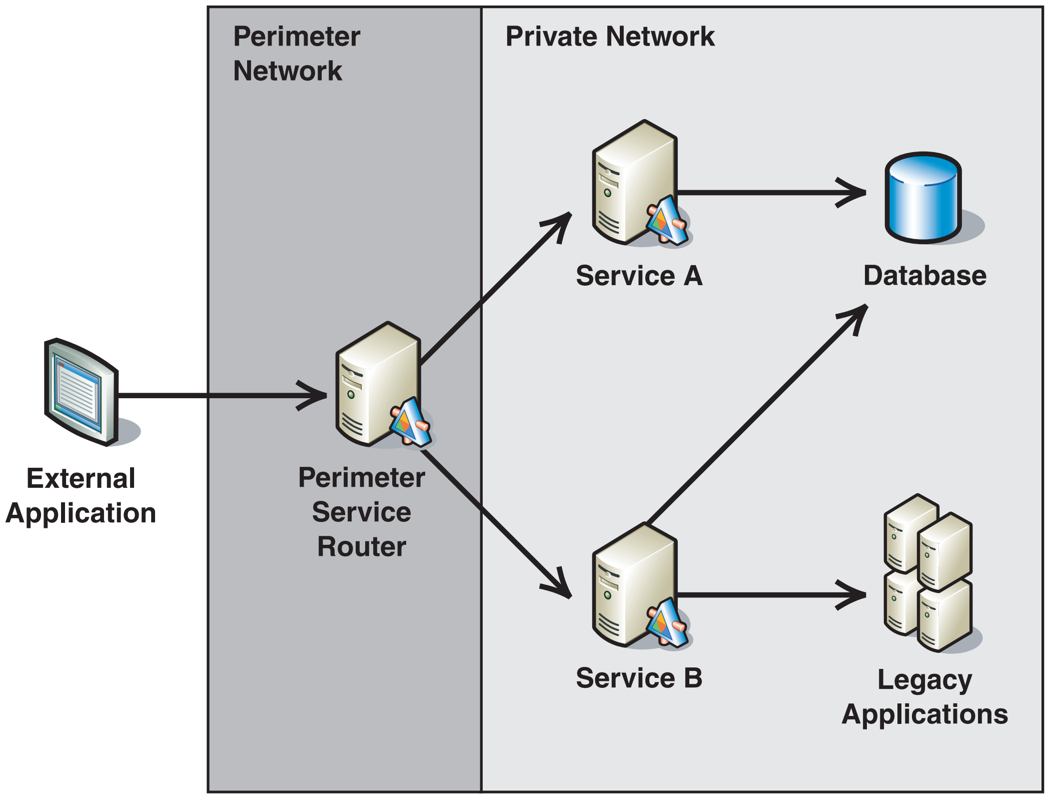


Figure 6.3
The deployment of a perimeter service router

Note: The code examples provided in this implementation pattern display the router and the service deployed on the same host for demonstration purposes only. Normally, you should deploy the perimeter service router to a server located on the perimeter network, and then deploy the service to a private network segment, as the previous diagram indicates.

Process

The Perimeter Service Router pattern provides a high-level overview of the perimeter service router functionality. This pattern describes the same process with refinements that are specific to this implementation. Figure 6.4 illustrates the functionality of the perimeter service router.

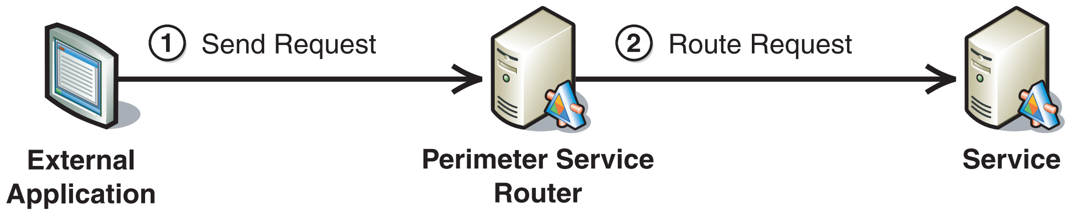


Figure 6.4
The functionality of a perimeter service router

The following steps show how the perimeter service router functions:

1. **The external application sends a request message**. The request message is addressed to the service's external interface as defined in the **<r:for>** entry in the referral cache on the perimeter service router.
2. **The perimeter service router forwards the request message to the service**. The perimeter service router directs the message to the target service URI defined in the **<r:go>** entry in its referral cache.

Implementation Approach

This section describes how to implement the pattern. The section is divided into four major tasks:

1. **General setup**. This task provides steps that apply to all applications for this pattern.
2. **Configure the external application**. This task lists the steps required to configure the external application to work with the perimeter service router.
3. **Configure the perimeter service router**. This task lists the steps required to configure policy and code on the perimeter service router.
4. **Configure the service**. This task lists the steps required to configure policy and code on the service to work with the perimeter service router.

This document describes the steps specific to implementing the perimeter service router. However, this document does not include details about how to implement authentication or message protection between the external application and service. For more information about authentication and securing communication between the external application and the service, see the following patterns:

- Direct Authentication in Chapter 1, "Authentication Patterns."
- Brokered Authentication in Chapter 1, "Authentication Patterns."
- Data Confidentiality in Chapter 2, "Message Protection Patterns."
- Data Origin Authentication in Chapter 2, "Message Protection Patterns."

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

General Setup

You must install WSE 3.0 on the computers that you use to develop WSE 3.0-enabled applications. Once WSE 3.0 is installed, you must enable the perimeter service router and the service to support WSE 3.0. You can achieve this by performing the following steps:

► To enable a Visual Studio 2005 project to support WSE 3.0

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **General** tab, select the checkboxes for the following options:
   a. **Enable this project for Web Services Enhancements**.
   b. **Enable Microsoft Web Services Enhancement SOAP Protocol Factory**
3. Click **OK**.

Configure the External Application

The external application requires no special configuration to use the perimeter service router in order to communicate with resources on the private network. However, the Web service publisher must create a copy of the service's WSDL file and change the URI to the perimeter service router's URI for external clients. The external copy of the WSDL file should contain all of the Web service operations that the router publicly exposes. This is the WSDL that the external application would use to generate its proxy to communicate with the Web service.

Configure the Perimeter Service Router

To configure the perimeter service router, you need to create an entry for a SOAP router in the perimeter service router's configuration file, and then specify the location of a referral cache. You can achieve this by performing the following steps.

► **To configure the perimeter service router**

1. In Visual Studio 2005, right-click the service router project, and select **WSE Settings 3.0**.
2. On the **Routing** tab, click **Add**.
3. In the **Type** drop-down list box, type **Microsoft.Web.Services3.Messaging.SoapHttpRouter, Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35**. This sets the class that WSE 3.0 uses to process messages for routing.

   Note: At the time that this document was published, the default value that was available in the drop-down list box, **Microsoft.Web.Services3.Routing.RoutingHandler, Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35** would not function properly. You should therefore use the value specified in Step 3 above.

4. In the path box, type the name of an external interface for the Web service, or if the service router will handle routing for many Web services, type "***.asmx**."

   In this pattern, the service is exposed through the router as **ExternalService.asmx**. This is reflected in the URI that the external application uses as the address for its request messages. For example, if the router is deployed to **http://perimeterserver/router/**, the external URI that the external applications use to communicate with the service through the router is: **http://perimeterserver/router/ExternalService.asmx**.
5. In the **Verb** drop-down list box, select * to route messages based on all verbs, and then click **OK**.
6. In the **Referral Cache** box, type a name for the referral cache, such as **referralCache.config**. For security reasons, name the referral cache with a .config suffix. For more information, see the Security Considerations section in this pattern.
7. Create a new referral cache file, or copy and modify an existing referral cache file, and then add it to the perimeter service router project.

Note: The account that the perimeter service router runs under must have read and write permissions for the referral cache file.

The following is an example of a routing referral cache for a perimeter service router.

```
<?xml version="1.0" ?>
<!-- This is the referral cache file that forwards calls through the router to a
service -->
<r:referrals xmlns:r="http://schemas.xmlsoap.org/ws/2001/10/referral">
   <r:ref>
      <r:for>

<r:exact>http://localhost/PerimeterServiceRouter/Router/ExternalService.asmx</r:ex
act>
      </r:for>
      <r:if />
      <r:go>

<r:via>http://localhost/PerimeterServiceRouter/Service/InternalService.asmx</r:via>
      </r:go>
      <r:refId>uuid:093DC599-FD40-4bd3-B15F-02698D8EBFC2</r:refId>
   </r:ref>
</r:referrals>
```

The URI specified in the previous code sample for the **<r:for>** element is the URI for the perimeter service router. The **<r:go>** element contains the URI for the service to which requests are routed. The **<r: via>** element specifies a URI to reroute the SOAP message. When there are multiple **<r: via>** elements, the SOAP request is routed only to the first **<r: via>** element. For more information on referral cache syntax, see How to: Configure the WSE SOAP Router.

Once the routing handler is configured, you can see an entry for an HTTP handler in the perimeter service router's configuration file that should look like the information in the following code sample.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 ...
 <system.web>
  <httpHandlers>
<add type=" Microsoft.Web.Services3.Messaging.SoapHttpRouter,
Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" path="ExternalService.asmx" verb="*" />
    ...
  </httpHandlers>
  ...
  </system.web>
  ...
</configuration>
```

Configure the Service

Web services that are not WSE-3.0 enabled will normally accept routed messages from a perimeter service router. However, a WSE 3.0-enabled Web service will reject a message that is not specifically addressed to it, unless it is configured to accept a message addressed to a different party. Because external applications use the external URI for the service on the perimeter service router to address request messages, you must configure a WSE-3.0 enabled service to accept messages that are addressed to the perimeter service router. You can do this by adding a **SoapActor** attribute above the class declaration code for your Web service class, as defined in the following code sample.

```
...
using Microsoft.Web.Services3.Messaging;
...
[SoapActor("http://localhost/PerimeterServiceRouter/Router/ExternalService.asmx")]
```

Substitute the URI in this code sample for the one that you use to externally expose your service on the perimeter service router.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Implementing Perimeter Service Router in WSE 3.0 pattern include the following:

- You can use the perimeter service router to extend the service boundary to the perimeter of the private network, which allows you to consolidate common perimeter security functions on the perimeter service router. For more information about this topic, see the Extensions section.
- You can take servers that host internal Web services offline for maintenance without affecting the external interface. You can accomplish this by configuring the perimeter service router to route messages to an alternate server while the staff performs maintenance on the primary server.

Liabilities

The liabilities associated with the Implementing Perimeter Service Router in WSE 3.0 pattern include the following:

- Each additional service interface that a service provides increases the amount of work required to change the functionality exposed by the perimeter service router.
- The implementation may add complexity and performance overhead that may not be justified for simple service-oriented applications.
- Internet Information Services (IIS) 6.0 locks the routing referral cache file for a deployed perimeter service router, which prevents direct modification of the cache. If you attempt to make modifications directly to the referral cache, you must first restart IIS 6.0 before you can save them. This requirement may affect the availability of the service router or other Web applications that the server may host. To resolve this issue, you can create another referral cache file, and then update the perimeter service router's Web.config file to point to the new referral cache file.

Security Considerations

Security considerations associated with the Implementing Perimeter Service Router in WSE 3.0 pattern include the following:

- External interfaces such as perimeter service routers are typically prime targets for attackers that represent major entry points to the private network.
- You should name your referral cache file with a .config extension. IIS 6.0 filtering prevents clients from accessing the contents in .config files. If you name the referral cache with a different extension, the filtering may not work properly and a client could access the contents to expose the internal URI of the Web service.

Extensions

This section discusses an extension that you can use to increase the functionality of the perimeter service router.

Extension 1 — Using the Perimeter Service Router as a Policy Enforcer

You can extend the perimeter service router to perform additional security functions by implementing a custom service router class that extends the **SoapHttpRouter** class. The custom service router can act as a policy enforcer to authenticate clients, perform message validation, reject replayed messages, attribute activity to a specific user or organization, and perform other security functions.

To use a custom service router class, the routing handler you create (that is described in the Implementation Approach section) must implement a custom router class that you can derive from the default **SoapHttpRouter** class.

You can override the following methods in the custom service router:

- **GetRequestPolicy**. This method allows you to implement logic that dynamically determines which policy is enforced for any request message received from the external application. This capability is useful for implementing message validation, rejecting replayed messages, and authenticating the caller.
- **GetForwardRequestPolicy**. This method allows you to implement logic that dynamically determines which policy is enforced for request messages passed from the perimeter service router to the service. This capability is useful for specifying policies for the perimeter service router to sign and encrypt an incoming request message from the external application.
- **ProcessRequestMessage**. This method allows you to implement logic that dynamically routes an incoming request message based on message content or other factors. For example, you can use this method to route an incoming message to an alternate destination while the primary recipient is offline and unable to accept request messages. When you override the **ProcessRequestMessage** method, the service might not use a referral cache unless it calls the **ProcessRequestMessage** method of the parent **SoapHttpRouter** class.

You can deploy the perimeter service router as an entry point for a trusted subsystem, which means that the service authenticates the routed request message, based on the perimeter service router's credentials instead of the original caller's.

In a trusted subsystem model, if you need to forward security claims from the original caller in the routed message, you must create a custom filter to add the claims to the security header of the request message. For more information about trusted subsystems, see Trusted Subsystem in Chapter 4, "Resource Access Patterns."

If you need to retain the security context of the original caller while doing anything other than simple pass-through routing as described in the base pattern, the external application must add claims to the request message to satisfy policy requirements for the internal service. In this case, the perimeter service router adds its own claims to the request message in addition to those that the requestor added. It then forwards the message to the internal service. The internal service then processes claims on request for both the external application, as the originating client, and the router to provide assurance that the message has passed through the router. For details on this approach, see the "SecureRoutingToUltimateReceiver" QuickStart sample in the WSE 3.0 QuickStarts folder.

For more information about implementing SOAP routers in WSE 3.0, see: Routing SOAP Messages with WSE.

More Information

"Service Interface Pattern" in *Enterprise Solution Patterns Using Microsoft .NET* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesServiceInterface.asp*.

For more information about using the WseWsdl3.exe utility, see the "WSDL to Proxy Class Tool" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/fbefe453-3851-439b-9c10-fb036b59ff81.asp*.

For more information on referral cache syntax, see "How to: Configure the WSE SOAP Router" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/6414f229-cead-48af-a293-cb893c24c0e6.asp*.

For more information about implementing SOAP routers in WSE 3.0, see: "Routing SOAP Messages with WSE" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/b41230fb-d0e1-48b1-88c0-3daf7a40c9e8.asp*.

7

Technical Supplements

Introduction

This chapter contains technical supplements for Kerberos and X.509 brokered authentication patterns. You can use these supplements in addition to the design and implementation patterns for their respective technologies. The supplements include specific guidance that may not directly relate to each design or implementation pattern, but they are likely to be important resources as you consider deploying a solution into production.

The Kerberos Technical Supplement for Windows includes:

- In-depth detail about how the Kerberos version 5 protocol is implemented on Windows Server 2003, including information on topics such as Local Security Authority (LSA), Security Support Provider Interface (SSPI), and key management.
- Definition and configuration of service accounts for Web services.
- Configuration of service principal names (SPNs) for use with Windows integrated authentication and message layer security.
- Kerberos operations for Web services that include the configuration of domain accounts and deploying Web farms using message layer security.
- Troubleshooting common Kerberos issues.

The X.509 Technical Supplement includes:

- An overview of public key cryptography, X.509 certificates, and digital signatures.
- Various uses of X.509 certificates to provide security.
- An overview of certificate authorities and certificate revocation.
- Information about how to obtain an X.509 certificate.

Kerberos Technical Supplement for Windows

The Kerberos version 5 protocol represents a network-based authentication service that uses tickets as a proof of identity. In most cases, users are still required to present a user name and password for authentication. However, after users are authenticated, they are issued a security ticket that is used to access protected resources. In contrast, with NTLM authentication, a hashed copy of the user's password is used to perform challenge/response authentication for each protected resource that the person wants to access.

The Brokered Authentication: Kerberos design pattern in Chapter 1, "Authentication Patterns" provides a step-by-step description of the Kerberos authentication process, along with benefits and liabilities associated with using the Kerberos protocol. Understanding the Kerberos authentication process is important, but it is also very helpful to understand how that process is implemented. As a result, this supplement focuses on the implementation of the Kerberos protocol on the Windows platform. It also discusses topics that relate to Web service implementations.

Local Security Authority (LSA)

The LSA Subsystem Service (LSASS) is the security subsystem in Windows that is responsible for:

- User authentication.
- Local system security policy, which controls who can log on to the computer, password policies, privileges that are granted to users and groups, and the system security auditing settings.
- Sending security audit messages to the event log.

User authentication in the LSASS is performed with security packages that are dynamically loaded at run time. There are two basic types of security packages; one is an authentication package that is accessed through a set of APIs, which are referred to as the LSA API. The other is named Security Support Provider (SSP), which is accessed through the Security Support Provider Interface (SSPI).

The LSA API is used for local authentication on a workstation or server. This API is called when you enter a user name and password at the CTRL+ALT+DEL login prompt, or when you use the Win32 **LogonUser** function that is available through the advapi32dll.

In Microsoft Windows NT® and Windows 2000, users must have Trusted Computing Base (TCB) privileges to use the **LogonUser** function. This is because it uses a low level LSA API function named **LsaLogonUser**, which requires system-level rights. In Windows XP and Windows Server 2003, **LogonUser** was modified so that TCB or system-level rights are not required. This function is not available in Windows 95, Windows 98, and Windows Millennium Edition.

Accessing the LSA

Most of the LSA API functions used for authentication and security context management require system-level privileges. Windows NT and Windows 2000 allowed users with TCB privileges to access these functions. However, in the Windows XP and Windows Server 2003 operating systems, a process must execute under the SYSTEM identity to access these functions. The reason for this restriction is that these functions have access to confidential information, such as the user's hashed password, which should never be accessible outside of the system.

Note: The original version of **KerberosToken** in Web Service Enhancements (WSE) 1.0 and WSE 2.0 used the LSA API directly. Processes that used this token required either TCB or SYSTEM privileges, depending on the operating system. There were also issues related to signing and encryption when the original version of **KerberosToken** was used in Windows 2000. For these reasons, you should use **KerberosToken2** in WSE 2.0 or **KerberosToken** in WSE 3.0, which both use SSPI.

When it comes to using the SSPI interface, it is not necessary for a process to run under the SYSTEM identity, or to have TCB privileges to perform authentication operations. This is good news for the Kerberos protocol because it means that applications that use the Kerberos SSP are not required to use a high-privilege level when they perform authentication. In other words, using TCB privileges or forcing a process to run as SYSTEM represents a significant security risk, which you can mitigate by using SSPI.

Security Support Provider Interface (SSPI)

SSPI defines a programming interface that security support providers (SSPs) must implement. Microsoft provides the following SSPs:

- Negotiate
- NTLM authentication
- Kerberos protocol
- Digest
- Secure Channel, such as transport layer security (TLS) and Secure Sockets Layer (SSL).

Negotiate is the preferred SSP for application developers to use because it attempts to first use the Kerberos protocol; if a Kerberos Key Distribution Center (KDC) is not available, it uses NTLM authentication. The Digest and Secure Channel SSPs are out of scope for this document.

To use SSPI, the first task is to load the desired SSP and access its security interface, which provides a table with function pointers to the appropriate SSPI operations. An application uses these function pointers to interact with the security support providers. The end result is that an application does not need to bind to a specific support provider. Instead, the operations are dynamically accessed through a function table.

SSPI is generally compatible with the Generic Security Services Application Programming Interface (GSSAPI), which has been published as Internet protocol specification (RFC 2743). GSSAPI represents a standard protocol that provides interoperability with other platforms that use the Kerberos protocol.

Note: Even though the GSSAPI has been implemented by the Kerberos SSP, there are some compatibility issues that you should consider. For more information about GSSAPI interoperability, see SSPI/Kerberos Interoperability with GSSAPI on MSDN.

The following section discusses important concepts that you should understand before learning the details of how applications use the SSP Interface for the Kerberos protocol.

Important Concepts

There are several concepts that are important to understand about implementing and using the Kerberos protocol. Because the Kerberos protocol is based on the use of shared secrets, understanding how shared secrets are created and accessed is very helpful. Other concepts that you should understand include service principal names (SPNs), and how the authenticator is used.

Shared Secrets

A main concept of the Kerberos protocol is how shared secrets are created and used for authentication. Essentially, a shared secret is nothing more than an encryption key that the Kerberos protocol uses to perform symmetric encryption. Symmetric encryption uses the same key to encrypt and to decrypt data. The Kerberos protocol uses several symmetric keys to encrypt different pieces of information as part of the authentication process. In addition, the Kerberos protocol supplies an encryption key that applications and services can use to sign and encrypt messages.

There are two main patterns related to using shared secrets: one uses a shared long term key to encrypt data, and the other uses long term keys and session keys. The following is a brief description of each pattern:

- A user's long term key is used to encrypt preauthentication data, which is decrypted by the domain controller that uses the same key.
- A service's long term key is used to encrypt a ticket, which contains a session key along with additional data. The session key is used to encrypt the authenticator. Both the ticket and authenticator are sent in a message. The receiving service uses its long term key to extract the session key and validate the authenticator.

Note: There are two types of tickets used by the Kerberos protocol: a ticket-granting ticket (TGT) used to access the ticket-granting service (TGS) and a service ticket used to access a service. Both keys are discussed later in this technical supplement.

Long Term Keys

Long term keys are stored in the credential store on a domain controller and in the credential cache of the LSA. Because these keys are associated with credentials, access to them is highly restricted, which means that access is limited to operations that run within the process of the LSA. In other words, client and server applications do not have access to the actual keys. Instead, LSA API operations use these keys to perform security operations.

Note: Long term keys are also referred to as master keys in many Kerberos protocol documents. When you see the term master key used in a document, remember it is referring to one of the long term keys in the following list.

The Kerberos protocol uses four long term keys to perform authentication:

- **User keys**. When a user is created, the user's password is used to create the user key. In Active Directory service domains, the user key is stored with the user's object in Active Directory. At the workstation, the user key is derived from the password when the user logs on.
- **Service keys**. Services use a key based on the password of the Windows account assigned to the process that is hosting the service. Typically this account is the host computer account. However, the process that is hosting a service can also be configured to use a user account.

  All KDCs in the same realm use the same service key, which is based on the password assigned to the krbtgt account. The krbtgt account is a disabled Windows user account that is created when an Active Directory domain is created.
- **System keys**. When a workstation or a server joins a Windows domain, a new computer account is created and a password is automatically generated. In the same manner as a user account, the computer account's password is used to create the system key.
- **Inter-realm keys**. To support cross-realm authentication, KDCs share an inter-realm key, which is the basis for transitive trust between domains.

As previously mentioned, a user or computer's password is converted into a long term key that is used for authentication with the Kerberos protocol. This is accomplished by performing a one-way hash on the plaintext password to create a cryptographic key. The default hash implementation in Windows creates a 128-bit key to support the RC4-HMAC encryption type.

When a user or computer account is created in Active Directory, the long term key is stored in the Security Accounts Manager (SAM) accounts database on the domain controller. This database is normally backed up, which means that offline attacks could be used to gain access to long term keys. As a result, a system key that is managed by administrators is used to encrypt the long term keys that are stored in the SAM.

When a user logs on to Windows, the plaintext password is converted into a cryptographic key and is immediately discarded. The cryptographic key, which is also the long term key, is then stored in a volatile memory-based credential cache on the local computer. If the password was typed correctly, the long term key should match the one stored in the SAM, which represents a shared secret between the local computer and domain controller.

Session Keys

Session keys are created for communication with the ticket-granting service (TGS) and the service. The Kerberos protocol uses the session key to encrypt an authenticator. The authenticator contains a timestamp and unique information that is used to authenticate the request. The authenticator also contains an optional sequence number field that can be used to provide message replay detection. Session keys are short term keys that can also be used to sign and encrypt messages.

Service Account

The term Service Account is used to describe the Windows account that a service uses when it performs operations with the Kerberos protocol. In other words, this is the account that the Kerberos protocol uses to retrieve a service's long term key from the credential cache. The actual account that is used is based on the application that is hosting the service and the configuration of that application. When a service ticket is requested from the TGS, the request must identify the service account so that the service ticket can be encrypted using the correct long term service key.

With Internet Information Services (IIS), the security configuration and IIS version plays an important role in determining what account will be used as the service account. Windows Integrated Security uses a different process than the one available when it implements the Kerberos protocol with message layer security. Also, IIS versions 5.*x* (which is used on Windows 2000 and Windows XP) has more limitations than IIS version 6.0 (which is used on Windows 2003).

Windows Integrated Security with IIS

When using Windows integrated security with IIS, the host computer account is used as the service account by default. With IIS version 6.0 you can override this behavior by creating a domain user account and using that account as the identity of the application pool that is used to host a Web service. This new account is the service account that a client application should use when it requests a service ticket from the TGS.

With IIS versions 5.*x*, Windows integrated security only uses the host computer account as the service account. This is important to remember because any client application that requests a service ticket for a service that is configured to use integrated security must use the service's host computer account as the service account in the request.

Message Layer Security with IIS

When you implement the Kerberos protocol with message layer security, the service account is based on the identity of the process that is used to host the Web service.

IIS version 6.0 uses an account named NETWORK SERVICE that has appropriate rights and which can be used for Kerberos authentication. With IIS 6.0 the identity of the application pool controls what account is used. By default this identity is set to NETWORK SERVICE. However, it is possible to use a domain user account or the SYSTEM account as the identity of the application pool.

With IIS versions 5.*x*, the default process account is ASPNET, which is a local account that does not have access to the network or the host computer account. To implement message layer security using the Kerberos protocol with IIS versions 5.*x*, you must modify the configuration of ASP.NET to use either the SYSTEM account or a domain user account as the process identity.

The “Kerberos Protocol Operations for Web Services“ section later in this technical supplement provides detailed instructions that you can use to create and configure a domain user account as the service account used by Web services.

Service Principal Names

An SPN is a unique identifier that applications can use to request a service ticket instead of using the service account name. The Kerberos protocol implementation in Windows uses the SPN to retrieve a valid service account from Active Directory. In other words, an SPN is another type of identifier that can be assigned to an account in Active Directory.

Without the use of an SPN, client applications that request service tickets must know the name of the Windows identity that is used as the service account to request a service ticket.

By using SPNs you do not need to expose account names and you have the ability to implement mutual authentication. In other words, a valid service response provides authentication that the service account associated with the SPN was used to process the request. As a result, when you request a service ticket, the use of SPNs is strongly recommended over the use of service account names.

SPN Types

There are two types of SPNs that can be created: one that is host-based and another that is arbitrary. When a new computer account is created in Active Directory, host-based SPNs are automatically generated for built-in services. Examples of these services include HOST, LDAP, and HTTP. In reality, SPNs are only created for the HOST service and all built-in services use the HOST SPN. However, this implementation is transparent because built-in names act as an alias to the HOST service unless they have been specifically mapped to a Windows account.

Note: The HOST service represents the host computer. The HOST SPN is used to access the host computer account whose long term key is used by the Kerberos protocol when it creates a service ticket.

The syntax that is used to identify a host-based SPN contains information about the computer that the service is running on and the port that it uses. The actual name is structured with the following syntax:

```
<ServiceClass>/<Host>:<Port>.
```

The following list describes each section of the name:

- ServiceClass is the service you are accessing, such as HTTP.
- Host is the computer name for the computer that hosts the service.
- Port is optional and only used for nonstandard port configurations.

Arbitrary SPNs use the following syntax:

```
<ServiceClass>/<ServiceName>.
```

As indicated in the previous example, this does not require the use of computer information.

The type of SPN that you use is based on the implementation of your service. With Web services, the same factors that affect what service account you can use also has an affect on the SPN that you use. For instance, when you use the host computer account as the service account, you should use a host-based SPN. When you use a domain user account as the service account, you should use an arbitrary SPN. However, in some cases, it may be necessary to define a host-based SPN that references a domain user account.

Tip: When you use Windows Integrated Security, both Internet Explorer and IIS use the HTTP SPN to request service tickets and to process a request. As a result, when you use a domain user account in IIS 6.0 as the process identity, you must map the host-based HTTP SPN to the domain account that is used by the service.

Service Classes

Service classes are arbitrary names that represent services, but are not linked to a specific service. Instead, the service class is nothing more than part of a unique key that is used to identify a service account. At the time of writing, a service class has not been defined for Web services. As a result, most examples use the HTTP service class. For instance, to access a service on a computer named London in a domain (a Kerberos protocol realm) named GLOBALBANK.net, the following SPN is used:

```
http/london.globalbank.net
```

The HTTP service class is one of the built-in services that act as an alias to the HOST SPN, which is mapped to the host computer account. This means that when you use the default HTTP service class, the Kerberos protocol uses the computer account as the service account when it requests a service ticket. This service class works well with the default configuration of IIS, regardless of which version you use. However, when you create new Web services, you should also create new service classes.

Because a service class is arbitrary, you can choose any name for a new service. However, it is not a good idea to use detailed names that are based on service actions, because the management of these names could become prohibitive. Instead, you should use a name that represents a suite of service actions, or even a suite of services.

When you configure constrained delegation in Windows Server 2003, the Service Type column contains the service class name. In other words, the service type that is used by constrained delegation is the same as a service class. As a result, service classes represent one of the primary identifiers that are used to control access with constrained delegation.

Defining an SPN

The tool that you use to create a new SPN is named setspn.exe. It can be found in the Windows Support Tools for Windows Server 2003.

► To create a new SPN

1. From the **Windows Support Tools** menu, open the command prompt.
2. Type the following syntax:

```
setspn -a <ServiceClass>/<Host | ServiceName> <ServiceAccount>
```

When you create a new service class for a host-based SPN, you need to create the following two new SPNs:

```
setspn -a AcmeService/LONDON LONDON
setspn -a AcmeService/LONDON.globalbank.net LONDON
```

These commands create a new service class for a host-based SPN that is mapped to the host computer account named **LONDON** using the Pre-Windows 2000 Domain Name System (DNS) name and a Fully Qualified Domain Name (FQDN). Both of these entries are required for host-based SPNs.

When you use a domain user account as the process identity of an IIS 6.0 application pool, you must map the HTTP SPN to the new domain account to use Windows Integrated Security. Use the following commands to create the HTTP SPN:

```
setspn -a HTTP/LONDON WS_Account
setspn -a HTTP/LONDON.globalbank.net WS_Account
```

With this example you are adding a new HTTP SPN to the Windows domain account named WS_Account. Because built-in service classes are just aliases to the HOST SPN, these commands create two new SPNs. However, it is important to understand that if HTTP SPNs have already been created you must first delete them before you can map the HTTP SPNs to another Windows account.

Use the following command to create an arbitrary SPN that maps to a specific domain user account:

```
setspn -a AcmeService/GlobalBank WS_Account
```

This command creates an arbitrary SPN named AcmeService/GlobalBank that maps to the WS_Account domain user account. You should also notice that the syntax is different. Instead of using a host and a domain after the service class, this uses a service name, which is also arbitrary. A detailed example that uses setspn.exe to map a domain account to a service that is running on Windows XP is discussed later in this chapter.

Kerberos Tickets

The Kerberos protocol specification uses the following description to define a ticket:

> *"A record that helps a client authenticate itself to a server; it contains the client's identity, a session key, a timestamp, and other information, all sealed using the server's secret key. It only serves to authenticate a client when presented along with a fresh Authenticator."*

This definition applies to both the ticket-granting ticket (TGT) and the service ticket, which were introduced in the discussion on shared secrets. Previous sections in this primer also describe how tickets are encrypted with a long term service key, which is the same as the "server's secret key" in the previous definition.

The most important concept to understand about Kerberos tickets is that tickets can only be accessed by the KDC and the service that a ticket was created for. This is because these are the only entities that have access to the service's long term key. Clients that use the Kerberos protocol do not have access to information that is contained in the ticket. However, clients do have a copy of the session key that is found in the ticket, which they use to encrypt the authenticator. A service authenticates the client by extracting the session key from the ticket and decrypting the authenticator.

Ticket Lifetimes

Each ticket issued by the Kerberos protocol has a start time and an expiration time. The ticket can be used as many times as necessary within that time frame. The Kerberos protocol itself does not keep records of tickets. This means that it is up to the ticket holder to renew tickets before they expire. If an expired ticket is presented to a service, an error is returned.

The recommended maximum lifetime for a ticket is one day. However, the default setting is 10 hours. This lifetime value is managed by Kerberos protocol policy settings. In addition, attributes on ticket-granting tickets (TGTs) can be used to enable the automatic renewal for a limited time to extend the lifetime of the TGT.

As previously mentioned, the TGT is used to request service tickets from a TGS. If the TGT is allowed to expire, it cannot be used to request a service ticket. On the other hand, service tickets are only used to authenticate new connections with a service. Ongoing operations are not interrupted if the service ticket expires during the connection.

Note: There is some inconsistency in how service tickets are described in many Kerberos protocol documents. Specifically there is a tendency to mix the names “service ticket” and “session ticket.” In other words, when you read “session ticket” think “service ticket.”

Something to keep in mind is that ticket lifetime is associated with tickets and not the authenticator, which is discussed in the next section.

Authenticator and Message Replay Detection

As mentioned previously, the Kerberos protocol uses two separate data structures to communicate with services. One is the actual ticket, which is created by either the authentication service (AS) for communication with the ticket-granting service (TGS), or by the TGS for communication with a service. The other structure is an authenticator. The authenticator contains two fields that can be used for message replay detection; one is the timestamp and the other is a sequence field.

A constraint imposed by the Kerberos protocol is that messages must be processed within a short time frame, usually five minutes, which is specified in the Kerberos protocol configuration. The way that the Kerberos protocol implements this behavior is through the use of a timestamp field on the authenticator. When a message is received, the authenticator is decrypted with the session key and the timestamp is examined to make sure it falls within the configured time frame on the target server. If it does not fall within the configured time frame, the message is rejected; this represents a type of message replay detection.

Another factor to consider is that the time on all computers within a realm must be synchronized for Kerberos protocol authentication to work properly. On the Windows platform, starting with Windows 2000, computers that are part of a domain are automatically synchronized with the domain controller.

Usually, relying on the timestamp for message replay does not protect against repeated messages that occur within the allowed time frame. As a result, the recommended approach for message replay detection is to use the sequence field in the authenticator. This is an optional field that is typically used for KERB_PRIV and KERB_SAFE messages, which are used to detect replays. This field can also contain a nonce (Number Once) value that can be cached by the service to implement replay detection. For more information about replay detection with the sequence field, see section "5.3.2 Authenticators" in RFC 1510, The Kerberos Network Authentication Service (V5).

Note: Windows does not use the authenticator's sequence field for message replay detection. Instead, Windows uses a message replay cache to implement replay detection. With this implementation, the timestamp is used as a unique value to protect against message replay attacks.

Delegation Configuration

Delegation on the Windows platform is implemented with the Kerberos authentication protocol. Windows 2000 Server supports unconstrained delegation while Windows Server 2003 supports both constrained and unconstrained delegation. The process of implementing delegation is beyond the scope of this chapter. However, it is important to understand basic requirements that relate to delegation configuration.

A key to correctly configure delegation is to understand what service accounts are being used by the application that attempts to perform the delegation, and what service is being called. For instance, suppose that you have an IIS Web application that attempts to access a Web service on a remote server using delegation. With delegation, the service account of the Web application is used to retrieve a service ticket to access the Web service. As a result the service account of the IIS Web application must be configured for delegation.

Delegation represents the ability of one service to request access to another service on behalf of a user. As a result, delegation is configured on the service account that requests access and not on the user account. The only exception is that the user account must be configured to support delegation. To do this, make sure the following account setting is not selected: **Account is sensitive and cannot be delegated**.

When using a domain user account as the service account, delegation must be configured on the domain user account and not on the computer account that hosts the service. To configure delegation in Windows Server 2003, you must first create an SPN for the domain user account. In other words, delegation options are not available on an account until that account is mapped to an SPN by using the setspn utility. On the Windows 2000 platform, delegation options are always available.

When you use constrained delegation, the service account of the application that is being called must be added to the list of allowed services in the delegation configuration dialog box on the service account that is implementing delegation.

For more information about constrained delegation, see Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."

The next section describes the detailed process that is performed when applications use the SSPI interface to perform Kerberos authentication.

Implementing Kerberos with SSPI

Figure 7.1 illustrates the main operations performed by applications that use the SSPI interface to implement the Kerberos authentication protocol between a client application and a Web service.

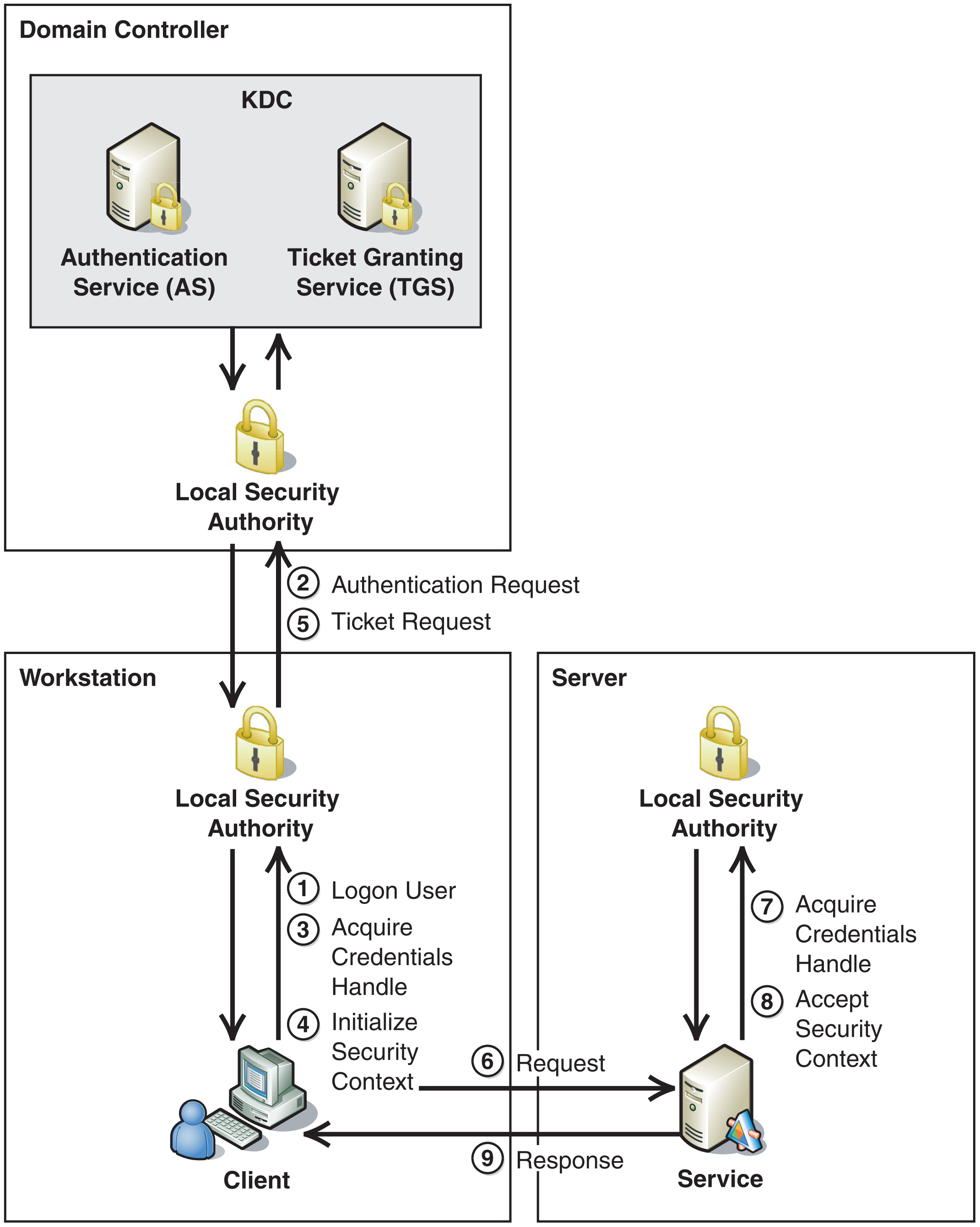


Figure 7.1
SSPI implementation of the Kerberos authentication protocol

As illustrated in Figure 7.1, SSPI implementation of the Kerberos authentication protocol is described in the following steps:

1. Initial logon is performed outside of SSPI. However, it is important to understand that a logon operation must be implemented prior to using SSPI. The actual logon process is based on the type of client application in use. For Windows applications the logon occurs when a user logs on to a workstation. For Web applications the logon occurs when a user accesses the Web site.

   It is also important to understand that after the logon is complete, all service operations should be performed with the identity of the user who has logged on to the system. The user can be either a person who interacts with an application or a business identity that is used for trusted subsystem implementations. Impersonation is usually required in Web applications so that SSPI will use the correct identity.
2. As previously mentioned, starting with Windows 2000 the default authentication provider is the Kerberos protocol on the Windows platform. As a result, the next operation is an authentication request that is sent to the LSA on the domain controller. After this operation completes, it returns a set of credentials that can be used to access a TGS. The credentials include a session key and a TGT, which are stored in a credential cache on the local computer.
3. The next step is to call the **AcquireCredentialsHandle** operation, which returns a handle to the credentials of the current user that was stored in the credential cache during the authentication request in the previous step.
4. The credentials handle is then used to initialize a security context that will be used to call a specific service. With the Kerberos protocol, the security context represents a data structure that contains credentials that are used to access a service, such as a session key and service ticket. To initialize the new security context, the LSA on the client workstation interacts with the LSA on a domain controller to request a service ticket from the TGS.
5. When performing a ticket request, the TGT and an authenticator that is encrypted with the session key are sent to the domain controller. The authenticator is a data structure that contains a timestamp along with other information, such as the Kerberos protocol version number. The LSA on the domain controller interacts with the TGS to obtain a new session key and service ticket, which are returned to the LSA on the client workstation.
6. After the security context is initialized, it is then used to access the Web service in the same manner that a request was made to the TGS when it obtained the service ticket. In other words, the session key that was returned in Step 5 is used to encrypt an authenticator, which is sent with the service ticket to the Web service.
7. When a service receives a message with a Kerberos service ticket and authenticator, the first step is to acquire the service's credentials, which are typically the credentials of the server process. These credentials contain the service's long term key that was used to encrypt the service ticket.

8. Both the Kerberos security context and the service's credentials handle are used in a call to **AcceptSecurityContext**, which is the operation that validates the service ticket that is contained in the message. Validation is performed by using the service's long term key to decrypt the service ticket and access the session key. The session key is used to decrypt the authenticator. Successful decryption of the authenticator is how the service authenticates the client. In addition, the timestamp included in the authenticator is used to limit the lifetime of the authenticator.
9. Optionally, a service can send a response to the client application.

Note: This description focuses on SSPI operations that are used to implement the Kerberos authentication process. For more information about Kerberos authentication, see Brokered Authentication: Kerberos in Chapter 1, "Authentication Patterns."

In addition to using the Kerberos protocol for authentication, you can also use Kerberos session keys for signing and encryption.

Signing and Encryption

As mentioned earlier, the Kerberos protocol security context contains a session key. The session key is a short-term symmetric encryption key used to encrypt the authenticator during authentication operations. This same session key can also be used by applications to implement XML signing and encryption. For more information about using symmetric keys for XML signatures and encryption, see Data Origin Authentication and Data Confidentiality in Chapter 2, "Message Protection Patterns."

With SSPI, you can choose from several methods that use the session key for signing and encryption. The following steps describe how to sign and encrypt a message that is sent from a client to a server.

Client:

1. Sign the message with **MakeSignature**.
2. Encrypt the message with **EncryptMessage**.

Server:

3. Decrypt the message with **DecryptMessage**.
4. Validate the message with **VerifySignature**.

Note: Starting with **KerberosToken2** in WSE 2.0, another SSPI function named **QueryContextAttributes** is used to access the session key directly for signing and encryption. Unfortunately, this operation is not available in Windows Server 2000. As a result, the only token in WSE 2.0, and earlier versions of WSE, that supports signing and encryption in Windows 2000 is **KerberosToken**.

Kerberos Protocol Operations for Web Services

This section provides information that you can use to perform different operations that relate to the Kerberos protocol and Web services.

Using a Domain Account with IIS 5.*x* (Windows 2000 and Windows XP)

Instead of using the default account that is defined in the **<ProcessModel/>** element of the Machine.config file, a service can use a domain user account as the process identity. The domain account needs additional privileges and if it is used for message layer security, an arbitrary SPN should be created.

► **To configure a domain account for the Kerberos protocol on a computer running IIS 5.*x*:**

1. Create a new user account in the domain (KDC realm) and add that account to the user group. This account does not need additional privileges on the domain computer. This means that you are using an account with the fewest privileges.
2. On the computer running IIS 5.*x*, the new domain account requires the following rights, which can be assigned with the Local Security Settings configuration tool:
   - **Log on as a service**
   - **Impersonate a client after authentication**
3. Assign **Full Control** permissions to the new domain account for the following folder on the IIS 5.x host:

   ```
   C:\%WINDOWS%\Microsoft.NET\Framework\v1.1.4322\Temporary ASP.NET Files
   ```

4. Update the <**ProcessModel**/> element in the Machine.config file on the computer that is running IIS 5.*x*. Both the user name and password need to be updated to values associated with the new domain account. Restart IIS.

   Note: The following step is required when you are using message layer security with the Kerberos protocol. When you use standard Windows authentication, it is not necessary to create an SPN for the account unless the account will be used for Delegation.

5. Use the setspn.exe tool to create an arbitrary SPN for the domain account. This action is performed on the Active Directory domain controller, not on the computer that is running IIS 5.*x*. To perform this action, you must be an administrator or have **SetPrincipalName** permissions on the domain controller. The following example creates an arbitrary SPN named **AcmeService/GlobalBank** that maps to a Windows account named **WS_Account**:

   ```
   setspn -a AcmeService/GlobalBank WS_Account
   ```

When you create a domain account that will be used for delegation with Windows Integrated Security, it should map to the HTTP host-based SPN. If you use message layer security with WSE 3.0, use an arbitrary SPN as previously described. Finally, the process model used by IIS 6.0 in Windows Server 2003 is very different from IIS 5.*x*. As a result, the steps previously described will not work in Windows Server 2003.

For step-by-step instructions on creating a domain user account and using the application pool identity in Windows Server 2003, see Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."

Web Farm Deployment with WSE 3.0

When you deploy Web services to a Web farm, you must use a domain account that maps to an arbitrary SPN for each Web server in the farm. When you configure services in a Windows 2000 Web farm, you can use the same technique that was previously described for using a domain account with IIS 5.*x*. When you configure services in a Windows 2003 Web farm, see Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resources Access Patterns," as a guide for using a domain account in that environment.

When you use the Kerberos protocol to access services in a Web farm, you must use the arbitrary SPN as the target principal, and not the HOST SPN. For example, when you use WSE 3.0, you use the arbitrary SPN as the **targetPrincipal** to initialize a **KerberosToken** security token.

Troubleshooting

This section contains information that you can use to troubleshoot common problems with the configuration and the implementation of the Kerberos version 5 protocol. If you are unable to resolve an issue after reading this section, see Troubleshooting Kerberos Delegation for in-depth troubleshooting information about the Kerberos protocol implementation in Windows 2000 and Windows 2003.

Duplicate SPNs

When creating and using new SPNs with Web services, you may need to perform some troubleshooting. For instance, if you accidentally map the same SPN to two different accounts, the SPN will no longer work. It may also be necessary to list all of the SPNs that are associated with an account to determine if a specific SPN has been created.

Windows Support Tools for Windows Server 2003 contains a utility named Ldifde.exe that you can use to list all accounts that map to a specific SPN.

▶ **To perform a query with Ldifde.exe**

1. From the **Windows Support Tools** menu, open the command prompt.
2. Type the following command:

```
ldifde -f c:\spn_out.txt -d "DC=globalbank,DC=net" -l serviceprincipalname -r
"(serviceprincipalname=HTTP/LONDON*)" -p subtree
```

This command searches for all of the SPNs in the **globalbank.net** domain that match the search mask **HTTP/LONDON***. It writes the results to a text file named **spn_out.txt** on drive C.

You can also use the setspn utility to list all of the SPNs that map to a specific account.

▶ **To list SPNs associated with an account**

1. From the **Windows Support Tools** menu, open the command prompt.
2. Type the following command:

```
setspn -l LONDON
```

This command lists all of the SPNs that map to the **LONDON** computer account. The default list should contain two HOST entries and if SQL Server is installed, it will contain an MSSQLSvc entry.

If you need to remove an SPN — for instance if you have mapped the HTTP SPNs to a domain user account and need to remove that mapping — you can use the following commands:

```
setspn -d HTTP/LONDON WS_Account
setspn -d HTTP/LONDON.globalbank.net WS_Account
```

Notice that this is the same syntax that you use to create an account, However, the command line option is **-d** (delete) instead of **-a** (add). Also notice that both the DNS and FQDN based names have been removed.

Cached Tickets

When a user or computer logs on to a domain with the Kerberos protocol, their credentials and account information are stored in a cache on the local computer. This cache only resides in memory and is not persisted to disk. However, it will remain active while the user or computer is logged in. The end result is that any ticket created by the Kerberos protocol for a user or computer is stored in the cache until the user or computer either logs out, or the cache is manually purged.

This behavior has the following significant impacts:

- Changes to account information, such as Active Directory group membership and delegation configuration, are not updated until the cache is cleared.
- Changes to SPN configuration, such as modifying the HTTP host-based SPN to use a different Windows account, are not picked up until the cache is cleared.
- Changing the password of a service account causes cached tickets that are assigned to the account to become invalid.

As a result of this caching, changes to the configuration may cause issues with the Kerberos protocol that are difficult to determine. The following two approaches can be used to purge the cache:

- Users can log off their computers and then log on again to purge user information, and the computers can be restarted to purge computer information.
- You can use tools in the Windows Resource Kit to purge tickets from the cache.

KerbTray and KList

The Windows Resource Kit provides two tools, named KerbTray and KList, which provide the ability to list and purge tickets that are used by the Kerberos protocol.

The Kerberos Tray (KerbTray) is a graphical user interface tool that displays ticket information for a computer that runs the Kerberos version 5 protocol. You can view and purge the ticket cache by using the KerbTray tool icon located in the system tray on the desktop. By positioning the cursor over the icon, you can see the time left on the initial TGT associated with the logon session before it expires. The icon also changes in the last hour before the LSA renews the ticket.

Kerberos List (KList) is a command line tool that can be used to list service tickets and initial TGT associated with the current logon session. KList also provides the ability to purge tickets associated with the current logon session.

Caution: Purging tickets used by the Kerberos protocol can impact functionality in the current logon session. In most cases, you can recover functionality by repeatedly attempting an operation. However; this method does not always work. If you have purged tickets and are not able to recover after several attempts, your only option is to log off and then log on again to the computer.

IIS Caching and Delegation

Another area that tends to cause problems when configuring delegation is the caching implementation in IIS. As previously mentioned, changes to delegation usually require purging tickets that are associated with the account that was modified. However, with IIS this behavior is different.

The following information describes how IIS behaves when you modify a service account to disable or enable delegation:

- When delegation is disabled on an account, IIS immediately picks up the change and denies access to downstream resources.
- When delegation is enabled on an account, IIS does not recognize the change. Instead, IIS must be restarted to pick up the delegation change and allow access to downstream resources.

For more information about configuring delegation, see the "Delegation Configuration" section earlier in this technical supplement.

X.509 Technical Supplement

The X.509 specification defines a standard for managing public keys through a Public Key Infrastructure (PKI). Public keys are maintained in X.509 certificates, which are digital documents that bind a subject's identity claims to a public key from a public/private asymmetric key pair. Identity claims are usually understandable by humans, such as a person's full name or e-mail address, or a machine host name or domain name. X.509 certificates are endorsed and issued by a trusted third party, which is known as a certificate authority (CA).

Public Key Encryption and Digital Signatures

Public key encryption, also known as asymmetric encryption, is based on a public/private key pair. The keys are mathematically linked, so that data encrypted with the public key can only be decrypted with the corresponding private key. X509 certificates use public key encryption as an alternative to shared symmetric keys, which are discussed in the Data Confidentiality pattern in Chapter 2, "Message Protection Patterns."

With public key encryption, the sender converts the plaintext message into ciphertext by encrypting it with the public key in the message recipient's X.509 certificate. The message recipient converts the ciphertext back into the plaintext message by decrypting it with the corresponding private key.

Figure 7.2 illustrates how public key encryption and decryption take place.

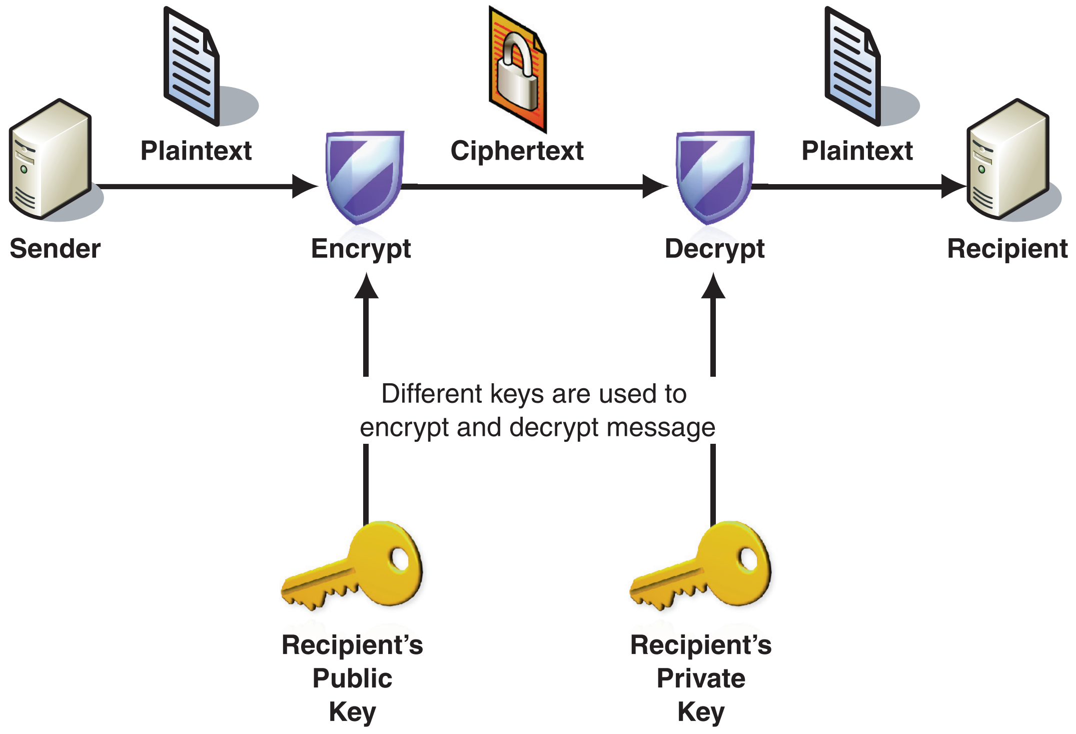


Figure 7.2
Public key data encryption and decryption

By using public key encryption, a message sender has assurance that only the recipient will be able to read the message.

In addition to providing data confidentiality through encryption, you can use the public key in the X.509 certificate to verify digital signatures created by a message sender. A digital signature is a value produced by the message sender to bind message data to the sender's identity and to provide a means of verifying the integrity of the message to detect tampering. In this case, the private key of the message sender is used to create the digital signature. The corresponding public key, which is found in the sender's X.509 certificate, is used to verify the signature. Digital signatures are used to assure the message recipient that the message originated from the identified sender, and that the message contents have not been altered since they were signed by the sender.

Note: With digital signatures that use public key cryptography, the origin of the signed message can be traced to the sender's identity, thereby satisfying nonrepudiation requirements. This differs from symmetric key integrity, where a message may have been signed by either party with knowledge of the shared secret key.

The public key can be distributed openly to encrypt messages and to verify digital signatures, but the private key in a key pair should be carefully guarded by its owner. This is necessary because it is used to prove the identity of the certificate subject and to decrypt messages that are intended for that subject.

Figure 7.3 illustrates the process of using public keys to sign a message.

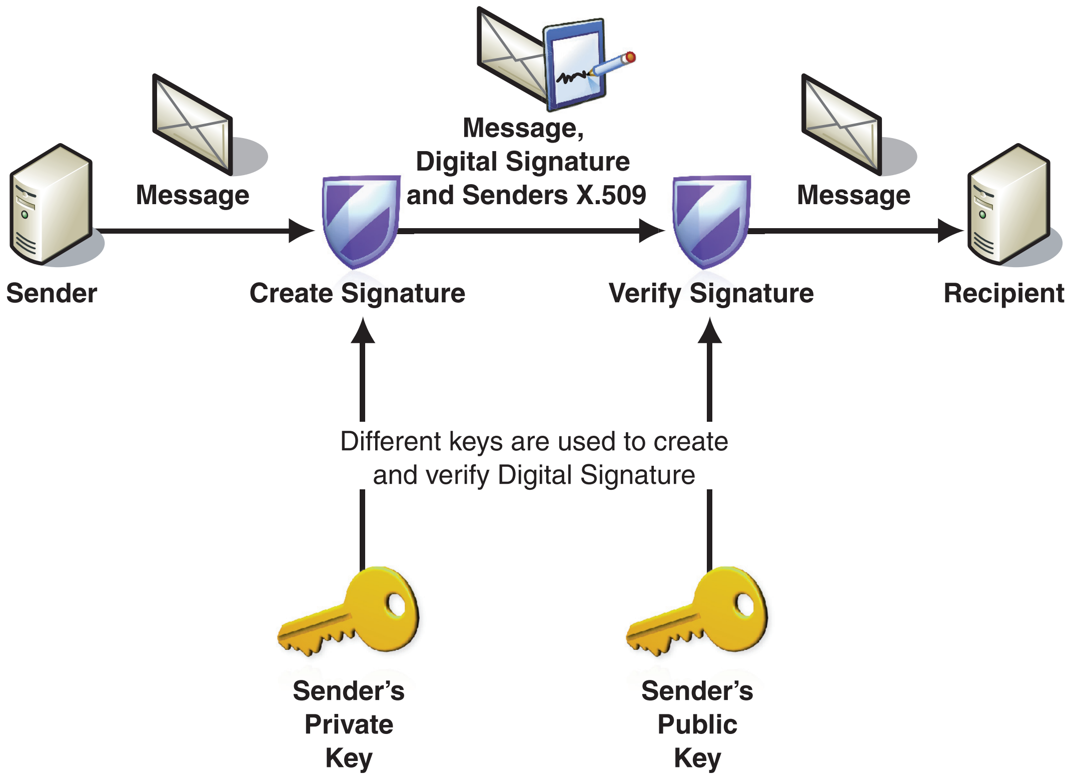


Figure 7.3
Creation and verification of a digital signature

For a more detailed description of data confidentiality, see the Data Confidentiality pattern. For more details about digital signatures, see the Data Origin Authentication pattern.

X.509 Certificates

X.509 certificates contain several required and optional attributes that enable the identification of the subject. You can obtain the following list of attributes in an X.509 certificate:

- **Version number**: The certificate version.

Note: Different versions (version 1, 2, and 3) of X.509 certificates have evolved over time, to provide additional security and attributes that are bound to the certificate. In practice, only version 3 certificates should now be used.

- **Serial number**: A unique identifier for the certificate.
- **Signature algorithm ID**: The algorithm used to create the digital signature.
- **Issuer name**: The name of the certificate issuer.
- **Validity period**: The period during which the certificate is valid. (This is typically set to be approximately one year.)
- **Subject name**: The name of the subject represented by the certificate. (The subject of a certificate is typically a person, an organization, or a Web/application server.)
- **Subject public key information**: The public key algorithm.
- **Issuer unique identifier**: The identifier for the issuer.
- **Subject unique identifier**: The identifier for the subject.
- **Extensions**: Extensions that can be used to store additional information. such as **KeyUsage** or **AlternativeNames**.
- **Signed hash of the certificate data**: The hash of the preceding fields encrypted using the issuer's private key, which results in a digital signature.

Custom security implementations that use X.509 certificates may depend on custom extensions that are not widely used or understood. These custom extensions must be included in the certificate by the certificate issuer when the certificate is created. Not all CAs may be willing or capable of adding custom extensions to certificates.

The validity period of an X.509 certificate tends to be much longer than that of other types of security tokens. For example, passwords are normally changed at shorter intervals, such as every 30 days. For this reason, it is critical to be aware of any possible compromise of an X.509 certificate private key, because it will be useful to an attacker for a considerably longer time than the secret key used in other security token types that have a much shorter lifespan.

Implementations of X.509

Security using X.509 certificates can be implemented at different layers of the network or application infrastructure, and each implementation had its own advantages and disadvantages.

Secure Sockets Layer (SSL)

SSL is a secure handshake protocol that supports X.509 certificates at the transport layer. It enables two parties to establish a session to communicate securely by providing confidentiality and data integrity. data origin authentication can also be provided if both parties use X.509 certificates. This is commonly referred to as SSL with client certificates. Some of the benefits of using SSL are:

- SSL is a well established protocol that is broadly interoperable, and is easy to configure and use.
- SSL has a performance advantage over message layer security because it is closer to the operating system than the message layer.

While SSL has some strong benefits, it does have the following liabilities:

- SSL operates point-to-point, which means that messages cannot be persisted in a secure state. It also means that SSL-encrypted SOAP messages cannot be processed by intermediaries without first being decrypted.
- If you use SSL in conjunction with WSE 2.0 or WSE 3.0 to provide data confidentiality and integrity at the transport layer, WSE cannot verify that SSL is being used to protect messages at the transport layer. Conversely, SSL cannot verify that clients are satisfying policy requirements defined in WSE, which is a requirement for client authentication.

WS-Security X.509 Binary Security Token

At the message layer, you can use X.509 certificates as binary security tokens in accordance with the WS-Security specification to sign and encrypt messages and to provide data confidentiality and data origin authentication.

The benefits of using X.509 at the message layer with binary security tokens include:

- Message layer security that uses X.509 certificates is flexible enough to provide point-to-point or end-to-end security. This allows messages to be persisted in a secure state for short periods for queue-based processing or for longer periods in an archived state.
- Message layer X.509 provides a high degree of interoperability. It provides standards based on the messages as they are sent over the wire instead of focusing on implementation for a particular platform.

Message layer security also has the following liabilities:

- Processing message layer security with X.509 certificates tends to have a greater impact on system performance than implementations that are lower in the protocol stack. This is because the message layer is further away from the hardware layer.
- Message layer security that uses X.509 certificates provides a great deal of flexibility, but it tends to be more complex to implement than security that uses X.509 certificates at other layers. This requires more knowledge of the underlying protocols, security policy, and programming against a Web services security API.

IPSec

IPSec provides a secure tunnel between two computers hosting applications that access resources or communicate with other applications. You can use X.509 in IPSec to authenticate hosts and negotiate a secure session between them. IPSec has some benefits that make it a viable security solution that uses X.509 certificates:

- **Performance**. IPSec benefits from better performance than security that is implemented further up the protocol stack, because it is closer to the hardware layer. It operates in the protocol stack between the data link and network layers.
- **Ease of configuration**. IPSec is easy to configure and implement on a number of platforms, including Windows Server 2003.

IPSec that uses X.509 certificates has a liability that must be considered:

- **No fine control of security**. IPSec policies are implemented based on a host computer instead of on a user or an application. IPSec that uses X.509 certificates is a viable option for providing secure communications between two hosts, but not for authenticating user or application subjects to make authorization decisions.

Certificate Authorities

Certificate authorities (CAs) are organizations that verify the identity of a subject that is represented in a certificate request, and that issue signed X.509 certificates. CAs can be internal or external to an organization. They can issue different types of certificates that are for a specific purpose or confer varying levels of trust.

External CAs are typically commercial entities that provide certificate issuance to customers for a fee. Examples of external CAs include Thawte, VeriSign, and RSA.

CAs offer different “grades” of signed certificates for purchase. Some have a nominal fee and come with minimal requirements to prove the subject’s identity. For example, a certificate that is used to sign e-mail messages may only cost a few dollars and require only e-mail confirmation to prove that the e-mail address represented by the subject in the certificate is authentic. A certificate that is used for more trusted activities may cost upwards of a hundred dollars and require a far more rigorous screening process to ensure that the subject meets the requirements for the certificate. Parties that want to use any type of certificate must decide that the criteria to qualify for such a certificate are sufficient for their needs, and that they consider the CA itself to be sufficiently reputable. The “grade,” “class,” or other term used by a CA to describe the quality or use of a certificate is often expressed as a certificate policy. A certificate policy describes the certificate’s applicability to a set of security requirements for a given purpose. For more information about certificate policies, see Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework.

Internal CAs, such as Certificate Services in Windows Server, can simplify certificate management activities, but in this case the trust of the certificate is now based on the organization that issued it. Certificates that are issued for subjects within the organization's security domain (usually defined in Active Directory) are typically signed with the organization's root certificate or another "parent certificate" that is allowed to sign certificates. For more information about X.509 PKI services on Windows Server 2003, see Designing a Public Key Infrastructure.

The chain of certificates, from the subject's certificate to the root certificate that is used by the CA for signing subject's certificate, is known as a trust chain. A party may decide to trust certificates at any level within the trust chain. This allows them to trust certificates further down the chain, as long as they are able to trace the trust chain back to the level of the certificate they original trusted.

Note: In test environments, you may choose to use certificates that do not have rigorous requirements for proving the identity of the subject. Certificates can be generated and self-signed with the MakeCert utility. However, there are known performance issues when verifying digital signatures with certificates that are generated by the MakeCert utility. For more information about the MakeCert utility, see Certificate Creation Tool (Makecert.exe).

Obtaining an X.509 Certificate

Depending on the type of CA, X.509 certificates can be obtained in a variety of ways. For external CAs, certificates are typically obtained for a subject by submitting a certificate signing request (CSR). A CSR contains the subject's name, the public key, and the algorithm that is used. (The majority of X.509 certificates you are likely to encounter use RSA for its algorithm).

The public key included in the CSR comes from a public/private key pair, which is generated specifically for use with the requested certificate. As soon as the public/private key pair is generated, the private key should be immediately stored in a secure place, such as a machine key store. Access to the key should be solely restricted to authorized parties. Ideally, the only party able to access the private key file is the subject that is represented in the X.509 certificate, although some infrastructures may allow access to the certificate private key by other accounts. When parties other than the subject represented by the X.509 certificate are allowed to access the certificate private key, the ability to support nonrepudiation may not be possible. The public key of the public/private key pair is required for the CSR, but the private key should never be sent to the CA under any circumstances.

Internal CAs may also use CSRs to process X.509 certificate requests. However, because the CA is internal to a specific organization, there can be additional options that reduce the overhead that is required to process requests and verify subject identities. For example, an internal CA that uses Windows Certificate Server may enable autoenrollment, which automates certificate request and issuance for user accounts that are created within an Active Directory domain. For more information about Public Key Infrastructure and Windows Server 2003, see Public Key Infrastructure for Windows Server 2003.

Figure 7.4 illustrates the process of a subject requesting and issuing an X.509 certificate with a CA that processes CSRs.

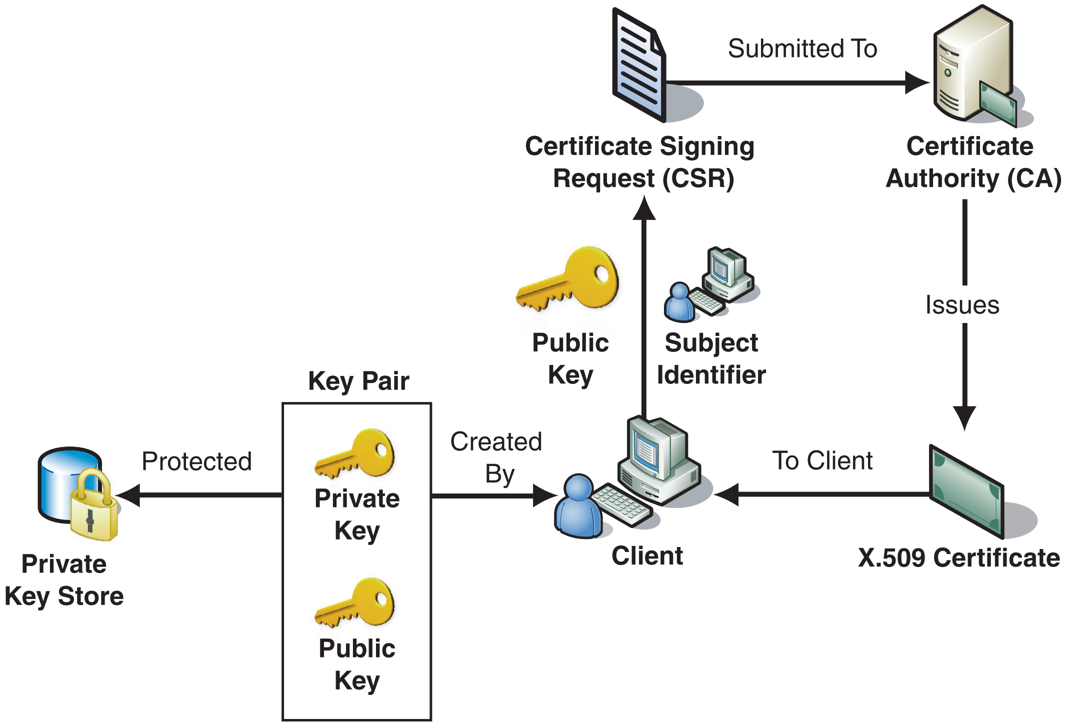


Figure 7.4

Requesting and obtaining a certificate from a CA

Certificate Revocation

The issuing CA can revoke X.509 certificates if the integrity of the certificate has somehow been compromised. The justification to revoke an issued certificate varies with each CA, but some general causes for certificate revocation include the following:

- **The private key has been stolen or wrongly disclosed due to improper storage or use**. For example, when the subject's private key is attached to an outgoing message instead of its X.509 certificate that contains the public key.
- **The subject represented in the X.509 certificate has breached the trust of the CA that issued the certificate**. For example, if information about the subject was intentionally misrepresented to the CA during the process of verifying the subject's identity.
- **An identity that corresponds to a certificate has been removed from an organization that manages an Internal CA**. For example, a user account is removed from the system or is disabled when the user's employment is terminated.
- **A subject no longer requires the certificate (cessation of operation)**. A CA may revoke a certificate if the certificate is no longer required and will not be used by the subject any more.

X.509 CAs typically publish a list of certificates that have been revoked, based on the CA's criteria for certificate revocation. These lists are known as certificate revocation lists (CRLs). CRLs are made publicly available so that a recipient can verify whether a certificate that was used to sign a message is valid. Any message recipient that receives a signed message should verify that the subject's certificate has not been revoked. This ensures the integrity of the signatures, based on the expected level of trust associated with the type of certificate.

In some situations, CAs may allow relying parties to query them directly to obtain the status of an X.509 certificate through an online revocation service (OLRS). A party that relies on this service communicates with the OLRS by using the Online Certificate Status Protocol (OCSP). If the CA offers access to an OLRS for the parties that rely on the service, it provides those parties with the ability to obtain the certificate status in real time instead of requiring them to download and cache CRLs published by the CA. One disadvantage of this approach is that it introduces a direct dependency upon the CA to be available to the parties that rely on it during the verification process. For more information about OCSP, see RFC 2650, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol — OCSP."

Certificate Storage and Access

X.509 certificates can be stored and accessed in a number of ways, including:

- **Local repository**. X.509 certificates may be exchanged out-of-band and stored in a locally accessible repository, such as the machine certificate store in the Windows operating system. You should only use a local repository if a small number of certificates are required for use by an online application.
- **PKI server**. Public Key Infrastructure (PKI) is a platform that allows an organization to centrally manage X.509 certificates that are required by the organization's services and subjects to authenticate and verify digital signatures. Certificates for that organization's subjects may also be made accessible outside of the organization. An example of a PKI solution is Certificate Services, which is included in Windows Server 2003. For more information about this PKI solution, see What Is Certificate Services?
- **Direct presentation**. X.509 certificates may be presented to a message recipient by attaching the certificate directly to the message. The recipient may subsequently decide to cache the certificate locally, pass it off to a central repository for storage, or simply reprocess it when it is attached to a new message.

Certificate Management

There are many issues related to certificate management and this section does not attempt to completely cover them. One issue that is significant to consider for message layer security is whether distinct certificates should be created for signing and encrypting message layer data.

For message-based security, it is a best practice to use distinct certificates and key pairs for encryption and digital signatures instead of a single key pair for both. One reason is that the contents of the certificates, as well as policies for issuance, key distribution, revocation, notification of revocation, and key backup are likely to differ depending on the purpose the keys are used for. This is particularly true when signatures are used for longer term authentication and integrity of business documents instead of merely temporary authentication of a session. Also, if encrypted messages are persisted to disk, you may need to decrypt the messages with an archived version of the private key. However, you do not want new digital signatures to be created with this private key.

Using X.509 Certificates in Patterns

Using X.509 certificates for authentication, data origin authentication, and data confidentiality is described in the following Web service security pattern documents:

- Brokered Authentication: X.509 PKI in Chapter 1, "Authentication Patterns."
- Implementing Message Layer Security with X.509 Certificates in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
- Implementing Transport Layer Security Using X.509 Certificates and HTTPS in Chapter 3, "Implementing Transport and Message Layer Security."
- Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

More Information

For information about compatibility issues between GSSAPI and the Kerberos SSP, see "SSPI/Kerberos Interoperability with GSSAPI" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthn/security/sspi_kerberos_interoperability_with_gssapi.asp*.

For information about replay detection with the sequence field, see section "5.3.2 Authenticators" in RFC 1510: *http://www.ietf.org/rfc/rfc1510.txt*.

For in-depth troubleshooting information for the Kerberos protocol implementation in Windows 2000 and Windows 2003, see "Troubleshooting Kerberos Delegation" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/tkerbdel.mspx*.

For information about Kerberos authentication, see "What Is Kerberos Authentication?" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/792ed95d-6f13-4181-a218-e4eaab361c1b.mspx*.

For information about certificate policies, see "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework": *http://www.ietf.org/rfc/rfc2527.txt*.

For information about X.509 PKI services on Windows Server 2003, see "Designing a Public Key Infrastructure" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/DepKit/b1ee9920-d7ef-4ce5-b63c-3661c72e0f0b.mspx*.

For information about the MakeCert utility, see "Certificate Creation Tool (Makecert.exe)" on MSDN: *http://winfx.msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_fxtools/html/b0343f8e-9c41-4852-a85c-f8a0c408cf0d.asp*.

For information about PKI and Windows Server 2003, see "Public Key Infrastructure for Windows Server 2003": *http://www.microsoft.com/windowsserver2003/technologies/pki/default.mspx*.

For information about the Online Certificate Status Protocol (OCSP), see RFC 2650, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol — OCSP": *http://www.ietf.org/rfc/rfc2560.txt*.

For information about the Certificate Services PKI solution in Windows Server 2003, see "What Is Certificate Services?": *http://www.microsoft.com/technet/prodtechnol /windowsserver2003/library/TechRef/63e3ba1c-cc23-40b1-9ca2-853869677318.mspx*.

For more information about certificates, see "What are certificates?" on the RSA Laboratories Web site: *http://www.rsasecurity.com/rsalabs/node.asp?id=2277*.

For information about Secure Sockets Layer (SSL), see "What is SSL?" on the RSA Laboratories' Web site: *http://www.rsasecurity.com/rsalabs/node.asp?id=2293*.

For more information about WS-Security version 1.0, see the OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/specs/index.php#wssv1.0*.

For information about IPSec, see "Internet Protocol Security (IPsec) Operations Topics": *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/operations /ipsec.mspx*

For information about the Internet X.509 PKI certificate and CRL profile, see "Internet X.509 Public Key Infrastructure Certificate and CRL Profile" (RFC 2459): *http://www.ietf.org/rfc/rfc2459.txt*.

Kaufman, C., Perlman, R., and Speciner, M. *Network Security — PRIVATE Communication in a PUBLIC World*. Upper Saddle River, NJ: Prentice Hall PTR., 2002, ISBN: 0130460192.

Appendix

Introduction

This section contains several topics that provide additional information related to the rest of the guidance:

- **Problem/Solution Index**. This index provides an alternative way to navigate the content in this guide that is based on frequently asked questions from customers. The index presents customer questions, and then directs you to the appropriate section of the guide to help you answer those questions.
- **WSE 3.0 Security: Interoperability Considerations**. This topic provides an overview of interoperability issues that you may encounter when developing Web services secured using SOAP message security.
- **Policy Advisor for WSE 3.0**. The Policy Advisor is a security tool for WSE 3.0 that you can use to help you review the security of WSE 3.0 installations. The tool examines the configuration and policy files for one or more WSE 3.0 endpoints, highlights typical security risks, and provides some remediation advice.
- **Patterns: A Common Vocabulary for Information Technology Professionals**. This white paper considers how patterns have influenced the Information Technology industry and looks forward to propose that patterns should become the basis for a common vocabulary among Information Technology (IT) professionals.
- **Glossary**. The Glossary contains a brief summary of key terms and definitions that appear in the *Web Service Security* guide.

Problem/Solution Index

During the research phase for the *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0* guide, the Microsoft patterns & practices team spent many hours communicating with customers, and collecting information from Microsoft Support Services, blogs, and other sources. This information helped the team gain a thorough understanding of the types of security challenges customers encountered when designing and implementing Web services using WSE 2.0.

The Problem/Solution Index provides an alternative way to navigate the content in this guide that is based on frequently asked questions from customers. The index presents customer questions, and then directs you to the appropriate section of the guide to help you answer those questions. The index is not comprehensive, but it does provide an alternative way to approach specific challenges. The index is divided into several broad categories to correspond to the areas where customers most frequently encounter problems.

The patterns & practices team hopes to expand the Problem/Solution Index as more questions related to the Web Service Security guide content emerge. You can submit additional questions to Web Service Security community workspace or add new problem/solution links to the Web Service Security Wiki.

General

For answers to general questions about WSE 3.0, see the resources in Table A.1.

Table A.1: General Questions

Problem	Solution
What is the difference between message and transport layer security?	See the Introduction in Chapter 3, "Implementing Transport and Message Layer Security."
How do I decide between message and transport layer security?	See the Introduction in Chapter 3, "Implementing Transport and Message Layer Security."
What interoperability considerations should I be aware of for WSE 3.0?	See WSE 3.0 Security: Interoperability Considerations in the "Appendix."

Authentication and Authorization

For answers to authentication and authorization questions, see the resources in Table A.2.

Table A.2: Authentication and Authorization Questions

Problem	Solution
How do I determine how to authenticate a client application?	See the Introduction in Chapter 1, "Authentication Patterns."
How do identification, authentication and authorization relate?	See the Introduction in Chapter 1, "Authentication Patterns."
How do I decide between Kerberos, X.509 or an STS based authentication broker?	See the Introduction in Chapter 1, "Authentication Patterns."

(continued)

Table A.2: Authentication and Authorization Questions *(continued)*

Problem	Solution
How can I obtain single sign on (SSO) within my intranet?	See the Introduction in Chapter 1, "Authentication Patterns."
How do I implement session-based authentication so that users are not required to provide their passwords whenever the application they are using calls a Web service?	See the Introduction in Chapter 1, "Authentication Patterns."
How can I use an existing Active Directory infrastructure for authentication?	See the Introduction in Chapter 1, "Authentication Patterns."
How do I provide authentication that is portable across organizational boundaries?	See the Introduction in Chapter 1, "Authentication Patterns."
How do I authenticate when interoperability is a challenge?	See the Introduction in Chapter 1, "Authentication Patterns."
How do I avoid using clear text passwords?	See the Introduction in Chapter 1, "Authentication Patterns."
How do I authenticate with **UsernameTokens** and secure the communication with X.509 certificates?	See Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I to authenticate against a directory service such as Active Directory or Active Directory Application Mode (ADAM) using a user ID and password?	See Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I authenticate against a custom SQL Server database, using a security token that contains a user ID and password?	See Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I develop a custom **UsernameTokenManager** to support authentication against ADAM or a custom SQL Server database?	See Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I make use of Visual Studio 2005 authentication services for SQL Server and a directory service?	See Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I implement mutual authentication using X.509 certificates?	See Implementing Direct Authentication with UsernameToken in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Kerberos Protocol and Windows Server 2003

For answers to questions about the Kerberos protocol and Windows Server 2003 in WSE 3.0, see the resources in Table A.3.

Table A.3: Kerberos Protocol and Windows Server 2003 Questions

Problem	Solution
How do I use an existing Kerberos protocol infrastructure at the message layer with a **KerberosToken** binary security token?	See Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I provide data confidentiality and data integrity to secure the communication channel by encrypting and signing the message with the **KerberosToken**?	See Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
How do I impersonate the client represented by the **KerberosToken** to access a resource on its behalf?	See Implementing Message Layer Security with Kerberos in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."
Is the Windows implementation of the Kerberos protocol compatible with other implementations?	See the Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."
How do I configure Active Directory for secure Web services using the Kerberos protocol in an implementation deployed in a Web farm?	See the Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."
How do I troubleshoot issues related to using the Kerberos protocol with Web services?	See the Kerberos Technical Supplement for Windows in Chapter 7, "Technical Supplements."

X.509 Certificates

For answers to questions about X.509 certificates in WSE 3.0, see the resources in Table A.4.

Table A.4: X.509 Certificate Questions

Problem	Solution
How do I create X.509 certificates?	See Brokered Authentication: X.509 PKI in Chapter 1, "Authentication Patterns," and the X.509 Technical Supplement in Chapter 7, "Technical Supplements."
How do I use X.509 certificate revocation?	See Brokered Authentication: X.509 PKI in Chapter 1, "Authentication Patterns," and the X.509 Technical Supplement in Chapter 7, "Technical Supplements."

(continued)

Table A.4: X.509 Certificate Questions *(continued)*

Problem	Solution
How do I authenticate users with X.509 certificates, and then perform role-based access control using an Active Directory domain?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I implement a custom WSE 3.0 X.509 **SecurityTokenManager** to allow additional data, such as roles to be associated with a user's certificate?	See Implementing Message Layer Security with X.509 Certificates in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Message Protection: Data Confidentiality, Integrity and Data Origin Authentication

For answers to questions about message protection in WSE 3.0, see the resources in Table A.5.

Table A.5: Message Protection Questions

Problem	Solution
How do I protect against eavesdropping or unauthorized access to data within a message?	See the Introduction in Chapter 2, "Message Protection Patterns."
How do I encrypt data within my message?	See the Introduction in Chapter 2, "Message Protection Patterns."
How do I protect against data tampering within a message?	See the Introduction in Chapter 2, "Message Protection Patterns."
How do I provide assurance to a message recipient that a message was sent by the expected sender?	See the Introduction in Chapter 2, "Message Protection Patterns."
How do I provide assurance to a message recipient that a message has not been altered after it was sent?	See the Introduction in Chapter 2, "Message Protection Patterns."
What is the difference between an XML signature and a digital signature?	See Data Origin Authentication in Chapter 2, "Message Protection Patterns."

Resource Access

For answers to questions about resource access in WSE 3.0, see the resources in Table A.6.

Table A.6: Resource Access Questions

Problem	Solution
What is the difference between impersonation and delegation?	See the Introduction in Chapter 4, "Resource Access Patterns."
How do I decide whether to use impersonation and delegation or the Trusted Subsystem model to secure access to resources?	See the Introduction in Chapter 4, "Resource Access Patterns."
How do I control access to a remote resource based on a user's identity instead of the identity of the application that is accessing the resource for the user?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I implement protocol transition on a computer running Windows Server 2003?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I implement impersonation?	See Implementing Message Layer Security with Kerberos in WSE 3.0 and Implementing Brokered Authentication Using Windows Integrated Security on IIS in the References for Transport Layer Security section in Chapter 3, "Implementing Transport and Message Layer Security."

Windows Server 2003 Protocol Transition and Constrained Delegation

For answers to questions about Windows Server 2003 Protocol Transition and Constrained Delegation in WSE 3.0, see the resources in Table A.7.

Table A.7: Windows Server 2003 Protocol Transition and Constrained Delegation Questions

Problem	Solution
How do I authenticate users with one protocol, and then authorize them to access resources using another protocol?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I use forms authentication on a presentation tier Web application, and then control access to back-end resources using Active Directory?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."

(continued)

Table A.7: Windows Server 2003 Protocol Transition and Constrained Delegation Questions *(continued)*

Problem	Solution
How do I authenticate users with X.509 certificates, and then perform role-based access control using an Active Directory domain?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I use protocol transition to initialize a **WindowsIdentity** object for authorization checks?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I use protocol transition to initialize a **WindowsIdentity** object for impersonation?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I use constrained delegation to access remote resources?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."
How do I create a service principal name (SPN)?	See the Protocol Transition with Constrained Delegation Technical Supplement in Chapter 4, "Resource Access Patterns."

Exception Shielding

For answers to questions about exception shielding in WSE 3.0, see the resources in Table A.8.

Table A.8: Exception Shielding Questions

Problem	Solution
How do I prevent my application from unintentionally disclosing sensitive information about itself through unhandled exceptions?	See Exception Shielding and Implementing Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."
How do I prevent the service from disclosing sensitive information in exception messages?	See Implementing Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."
How do I create exceptions that are safe by design containing information that I can return to Web service clients?	See Implementing Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."
How do I write unsanitized exception details to a log to support monitoring and troubleshooting?	See Implementing Exception Shielding in Chapter 5, "Service Boundary Protection Patterns."

Message Validation

For answers to questions about message validation in WSE 3.0, see the resources in Table A.9.

Table A.9: Message Validation Questions

Problem	Solution
How do I prevent a Web service from processing a message that contains malicious content?	See Message Validator and Implementing Message Validation in WSE 3.0 sections in Chapter 5, "Service Boundary Protection Patterns."
How do I reduce an attacker's ability to bring down my Web service with junk messages?	See Message Validator and Implementing Message Validation in WSE 3.0 sections in Chapter 5, "Service Boundary Protection Patterns."
How do I prevent the service from processing request messages that are greater in size than a specified limit?	See Implementing Message Validation in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."
How do I prevent the service from processing messages that are not formed correctly or that do not conform to an expected XML schema?	See Implementing Message Validation in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."
How do I validate input messages before deserializing them into Microsoft .NET Framework data types so that they can be interpreted as regular expressions?	See Implementing Message Validation in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."
How do I create a custom assertion on WSE 3.0?	See Implementing Message Validation in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."
What ASP.NET and WSE 3.0 configuration settings exist to limit usage of resources such as CPU?	See Implementing Message Validation in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."

Message Replay Detection

For answers to questions about message replay detection in WSE 3.0, see the resources in Table A.10.

Table A.10: Message Replay Detection Questions

Problem	Solution
How do I protect a Web service from an attacker replaying intercepted messages?	See Message Replay Detection in Chapter 5, "Service Boundary Protection Patterns."
How do I prevent the service from accepting and processing messages that have expired, after allowing for variable clock skew?	See Message Replay Detection in Chapter 5, "Service Boundary Protection Patterns."
How do I prevent the service from accepting and processing messages that have been replayed by attackers?	See Message Replay Detection in Chapter 5, "Service Boundary Protection Patterns."
How do I support preventing against replay attacks for Web services deployed in a web farm through the use of a database backed replay cache?	See Message Replay Detection in Chapter 5, "Service Boundary Protection Patterns."
How do I implement message replay detection using a WSE 3.0 custom assertion?	See Implementing Message Replay Detection in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."

Secure Conversation

For answers to questions about secure conversation in WSE 3.0, see the resources in Table A.11.

Table A.11: Secure Conversation Questions

Problem	Solution
How do I optimize secure communications between two parties?	See "Extension 2 — Web Service Federation" in Brokered Authentication: Security Token Service (STS) in Chapter 1, "Authentication Patterns."

Service Router

For answers to questions about the service router in WSE 3.0, see the resources in Table A.12.

Table A.12: Service Router Questions

Problem	**Solution**
How do I make internal Web services available to external clients?	See Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
How do I route SOAP messages to an alternate service when my primary service is down for maintenance?	See Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
How do I create a policy enforcer that performs security functions before a message reaches my Web service?	See Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
How do I minimize exposure of my Web services while providing access to them through controlled points?	See Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
How do I route SOAP messages based on their content?	See Perimeter Service Router in Chapter 6, "Service Deployment Patterns."
How do I configure and use the **SoapHttpRouter** class in WSE 3.0?	See Implementing Perimeter Service Router in WSE 3.0 in Chapter 6, "Service Deployment Patterns."

More Information

To submit additional questions related to this guidance, see the community workspace "Web Service Security: Scenarios, Patterns, and Implementation Guidance": *http://go.microsoft.com/fwlink/?LinkId=57044*.

To add new problem/solution links related to this guidance, see the "Web Service Security Wiki": *http://go.microsoft.com/fwlink/?LinkId=57051*.

WSE 3.0 Security: Interoperability Considerations

The *Web Service Security* guide provides detailed information about how to provide Web services security in your environment, including implementation patterns that use Web Services Enhancements (WSE) 3.0 to implement a set of core standards, such as XML, SOAP, Web Services Description Language (WSDL), and WS-Security. In many cases, your environment will include multiple platforms, so to successfully act on some of the guidance, you need to understand some important interoperability issues that arise in a Web services environment.

This appendix provides an overview of interoperability issues that you may encounter when developing Web services secured using SOAP message security. It is not intended to provide a detailed analysis of other areas of interoperability relating to technologies such as XSD, WSDL, or SOAP. For in-depth information about interoperability and Web service security, see *WS-I Basic Security Profile 1.0 Reference Implementation: Preview release for the .NET Framework version 1.1* on MSDN. There is also a WSE 3.0 Community Technical Preview (CTP) of this application available on the Microsoft WS-I Basic Security Profile community workspace.

Interoperability Between WSE 2.0, WSE 3.0, and WCF

This section details the degree of interoperability that is available between the different Microsoft platforms for Web Service development.

WSE 3.0 and the Windows Communication Framework (WCF)

Microsoft provides on the wire interoperability between WSE 3.0 and WCF (formally code-named "Indigo"). This allows messages sent from a client based on WSE 3.0 to be consumed by a WCF service, and vice versa. WCF includes two standard bindings for backward compatibility with ASMX Web services and WSE 3.0. The **BasicHttpBinding**, which does not incorporate message layer security, provides wire-level compatibility with ASMX Web services, while the **WsHttpBinding** is fully interoperable with WSE 3.0 (with a few minor configuration changes).

However, there will not be an automated mechanism to upgrade WSE 3.0 applications to run on WCF. The WS-I BSP Reference Implementation application was designed to reduce the burden of upgrading code. For more information, see Chapter 6, Designing Web Services for Interoperability and Resilience of *WS-I Basic Security Profile 1.0 Sample Application: Preview release for the .NET Framework version 1.0*.

WSE 3.0 and WSE 2.0

Because of changes in the underlying specifications for WS-Addressing, WS-Trust, and WS-SecureConversation, WSE 3.0-enabled applications do not interoperate with WSE 2.0-enabled applications. However, WSE 3.0 and WSE 2.0 client applications can run side-by-side with the .NET Framework 2.0. You can host WSE 3.0-enabled Web services and WSE 2.0-enabled Web services on the same computer, but they must be in separate virtual directories for ASP.NET or separate applications for Windows Forms applications.

It is theoretically possible to develop Web services using WSE 2.0 in such a way that they can interoperate with WSE 3.0 (and WCF) by using only a reduced set of specifications — specifically, SOAP 1.1, WSDL 1.0, and WS-Security 1.0. However, Microsoft does not support interoperability in this situation. The safest way to plan for interoperability with WCF is to upgrade the WSE 2.0 code to WSE 3.0.

Web Services Security Interoperability with Other Platforms

Interoperability issues can arise for many reasons, ranging from incorrect implementations of XSD data types to varying support for extensibility points, such as which algorithms are implemented. Complete coverage of these issues is outside the scope of this appendix, but it does discuss three major areas that may result in interoperability issues.

Support for Advanced Web Services Specifications

Web service specifications are intentionally designed to be composable. Because of this, there are more advanced capabilities, such as layering security on top of the fundamental specifications that relate to messaging. To achieve interoperability between different platforms, you should understand how each platform has implemented the underlying Web service specifications.

Web services specifications implemented in WSE 3.0 include: WS-Security 1.0 and 1.1, WS-Trust, WS-SecureConversation, WS-Addressing (08/2004 draft), SOAP Message Transmission Optimization Mechanism (MTOM), SOAP 1.1, and SOAP 1.2. For more information about the implemented specifications, see the WSE 3.0 documentation.

Figure A.1 illustrates an overview of the Web services specifications related to security.

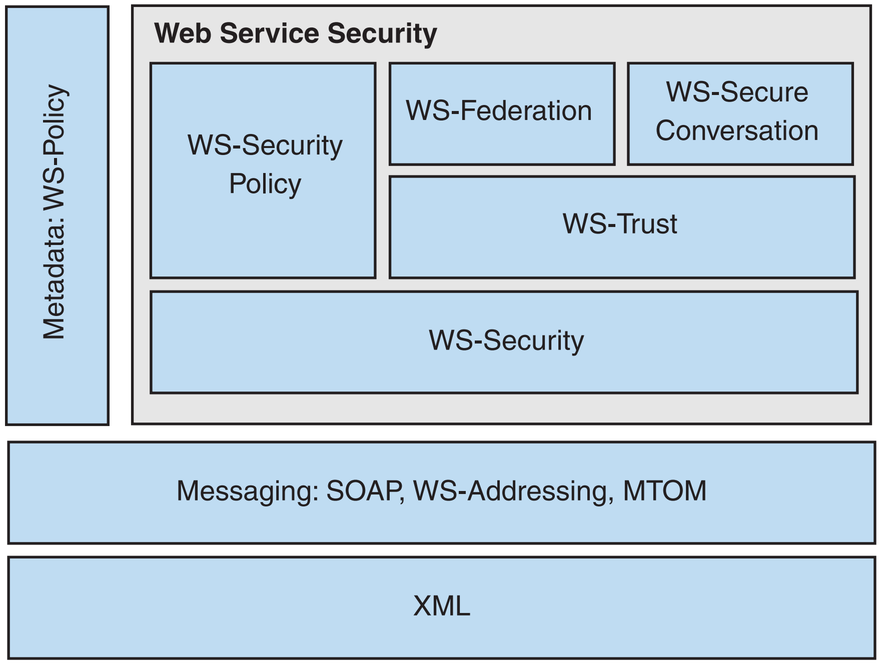


Figure A.1

The SOAP message security stack

Note that not all Web services specifications in Figure A.1 are currently implemented in WSE 3.0. Most major vendors currently support WS-Security 1.0, but some vendors may not yet have implemented more advanced specifications, such as WS-Trust and WS-SecureConversation.

Support for New Versions of Web Services Specifications

WSE 2.0 implemented WS-Security 1.0, but WSE 3.0 implements both WS-Security 1.0 (WSS1.0) and WS-Security 1.1 (WSS1.1). WS-Security 1.1 introduces several new capabilities that include:

- **XML digital signature confirmation**. Web services can now confirm to a client when an XML digital signature is verified. Clients can decide whether to accept SOAP responses from Web services that do not send signature confirmations.
- **EncryptedKey security tokens**. **EncryptedKeyToken** security tokens are used to optimize the performance of cryptographic operations when only the public key from an asymmetric key pair, such as a certificate, is present. You can use **EncryptedKeyToken** security tokens to secure SOAP message exchanges between anonymous clients that have only the public key for a Web service's certificate.

For a detailed description of each turnkey scenario, see the WSE 3.0 documentation. However, Table A.13 also provides a summary of WSE 3.0 turnkey scenarios, and their dependency on WS-Security specifications. If interoperability with WSS1.0 is required, make sure you use an assertion marked with an "X" in the WSS 1.0 column. You should ensure that your Web services support WSS 1.1 before requiring the client to check for the optional **SignatureConfirmation** capability because this is a WSS 1.1 feature. WSE 3.0 tooling automatically generates policy that has the **SignatureConfirmation** option set to **false**.

Table A.13: Summary of WSE 3.0 Turnkey Scenarios and WS-Security Dependence

WSE 3.0 Assertion	WSS 1.0	WSS 1.1	Features that require WSS 1.0
UsernameOverTransportSecurity	X		
UsernameForCertificateSecurity		X	**EncryptedKey** (required) **SignatureConfirmation** (optional)
MutualCertificate11		X	**EncryptedKey** (required) **SignatureConfirmation** (optional)
MutualCertificate10	X	X	**SignatureConfirmation** (optional)
KerberosSecurity	X	X	**SignatureConfirmation** (optional)
AnonymousForCertificateSecurity		X	**EncryptedKey** (required) **SignatureConfirmation** (optional)

The following code example shows an example of the **MutualCertifcate10** policy assertion with the **requireSignatureConfirmation** field, which you can be set to **true** or **false**.

```
<mutualCertificate10
  clientActor
  requireDerivedKeys="true|false"
  establishSecurityContext="true|false"
  messageProtectionOrder="Signature and encryption order"
  renewExpiredSecurityContext="true|false"
  serviceActor
  requireSignatureConfirmation="true|false"
  ttlInSeconds >
  <clientToken>
  <serviceToken>
  <protection>
</mutualCertificate10 >
```

Most major vendors currently support WS-Security 1.0, but some vendors may not yet have implemented WS-Security 1.1.

Varying Support for Extensibility Options Within the Specifications

A number of Web services specifications contain extensibility points that vendors may implement. For example, an extensibility point within the WS-Security specification is the selection of cryptography algorithms. WS-I has created profiles that help to reduce the number of options within extensibility points. For this reason, it is more likely that different platforms will interoperate.

One extensibility point used by WSE 3.0 and WCF that is not implemented by all vendors is the key transport algorithm. Key transport algorithms are used to optimize encryption by encrypting symmetric encryption keys, such as data encryption keys, with asymmetric encryption keys. WSE 3.0 and WCF use a default setting of RSA_OAEP. If you need to interoperate with an application that has not implemented RSA_OAEP, you may need to consider changing to RSA15 instead. RSA_OAEP is also not supported on Windows operating systems earlier than Windows XP. The following excerpt from the WS-I Basic Security Profile refers to the key transport algorithm.

R5621 When used for Key Transport, any xenc:EncryptionMethod/@Algorithm attribute in an **ENCRYPTED_KEY** MUST have a value of "http://www.w3.org/2001/04/xmlenc#rsa-1_5" or "http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"

The RSA (PKCS#1.5) algorithm ("http://www.w3.org/2001/04/xmlenc#rsa-1_5") is widely implemented and deployed in existing practice. The RSA-OAEP algorithm ("http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p") is relatively new and becoming widely implemented and deployed.

Example

The following configuration example specifies that symmetric session keys generated for **X509SecurityToken** security tokens are encrypted using the RSA15 algorithm instead of the default RSA_OAEP algorithm. This configuration is placed in the App.config file or the Web.config file of the server.

```
<configuration>
  <microsoft.web.services3>
    <security>
      <binarySecurityTokenManager>
        <add valueType="http://www.docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-x509-token-profile-1.0#X509v3"
        type="Microsoft.Web.Services3.Security.Tokens.X509SecurityTokenManager,
Microsoft.Web.Services3, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" priority="1" group="0">
          <!--Default value is keyAlgorithm name="RSAOAEP" -->
          <keyAlgorithm name="RSA15"/>
      </add>
    </security>
  </microsoft.web.services3>
</configuration>
```

If you encounter additional interoperability issues related to WSE 3.0, post a message on the Microsoft WS-I Basic Security Profile community workspace.

More Information

For in-depth information about interoperability and Web service security, see *WS-I Basic Security Profile 1.0 Reference Implementation: Preview release for the .NET Framework version 1.1* on MSDN: *http://msdn.microsoft.com/practices/guidetype/RefImp /default.aspx?pull=/library/en-us/dnpag2/html/MSWSIBSP.asp*.

For information about how the WS-I BSP Reference Implementation application reduces the burden of upgrading code, see Chapter 6, "Designing Web Services for Interoperability and Resilience" of *WS-I Basic Security Profile 1.0 Reference Implementation: Preview release for the .NET Framework version 1.0* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html /mswsibsp_chapter06.asp*.

For information about the WS-I Basic Security Profile 1.0 Reference Implementation application, see the WSE 3.0 Community Technical Preview (CTP) on the "Microsoft WS-I Basic Security Profile community workspace": *http://go.microsoft.com/fwlink/?linkid=47780&clcid=0x409*.

For information about the implemented specifications, see the "WSE 3.0 documentation": *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0 /html/d0ed7f06-504b-40f8-939c-b884ffce77c0.asp*.

For information about the WS-I Basic Security Profile, see "Basic Security Profile Version 1.0": *http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html*.

For information about SOAP Message Security 1.0, see "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) from OASIS": *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.

Policy Advisor for WSE 3.0

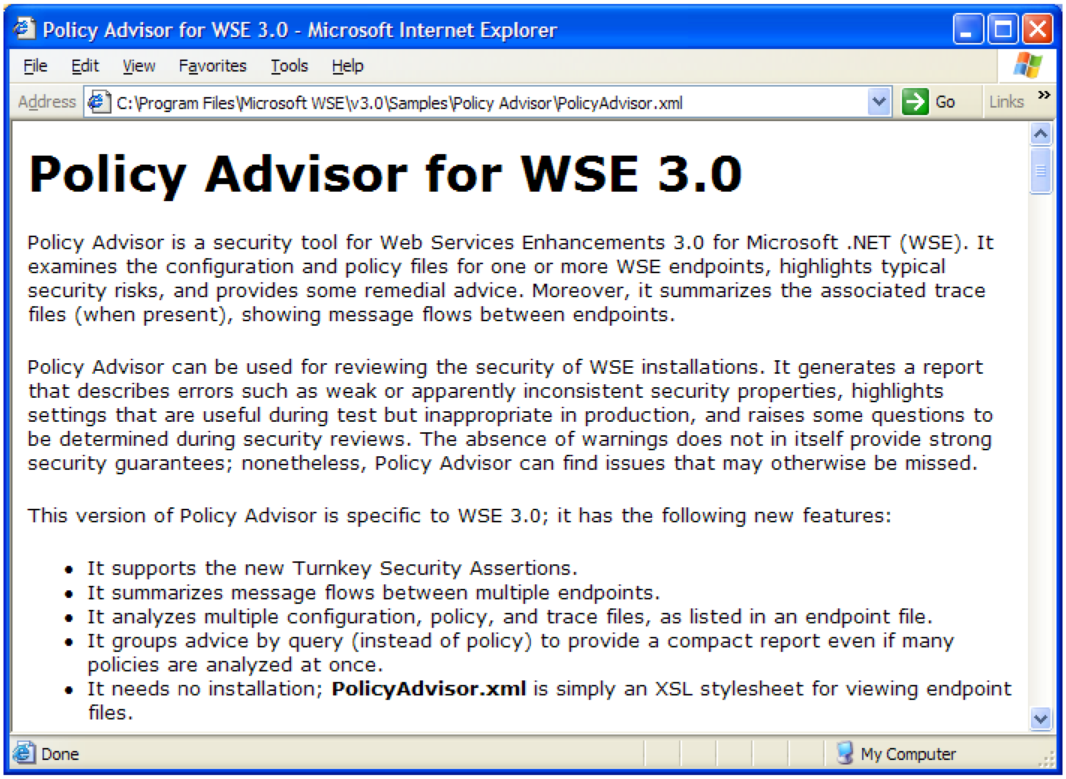
In Web services and clients implemented with WSE 3.0, you can use declarative XML configuration and policy files to determine many aspects of SOAP message processing. Separating security critical processing from code is considered good practice, because it makes it easier for manual review, and it allows you to customize during deployment without recompiling code. However, the flexibility of the configuration and policy formats creates a risk that subtle errors can occur. These errors can leave Web services vulnerable to replay, man-in-the-middle, redirection, and dictionary attacks. In the context of SOAP security, these are known as *XML rewriting attacks* to distinguish them from other types of attack, such as buffer overruns or SQL injections.

Policy Advisor is a security tool for WSE 3.0 that you can use to help you review the security of WSE 3.0 installations. The tool examines the configuration and policy files for one or more WSE 3.0 endpoints, highlights typical security risks, including XML rewriting attacks, and provides some remedial advice. The tool also summarizes the associated trace files when they are present, and displays message flows between the endpoints. Like most automated security tools, Policy Advisor can generate false alarms. Conversely, an absence of warnings does not guarantee an absence of security vulnerabilities. However, Policy Advisor isolates a range of vulnerabilities to XML rewriting attacks that you otherwise might not detect.

PolicyAdvisor.xml

Policy Advisor is implemented as an XSL transform that processes a user-supplied XML endpoints file to discover and analyze WSE 3.0 security policy and configuration files. After you install the samples, you can access the Policy Advisor tool in the WSE 3.0 installation at /samples/Policy Advisor/PolicyAdvisor.xml.

If you open the **PolicyAdvisor.xml** file in Internet Explorer, you can view the documentation for the Policy Advisor, including a list of all the security risks that the Advisor identifies, as shown in Figure A.2.


Policy Advisor for WSE 3.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address C:\Program Files\Microsoft WSE\v3.0\Samples\Policy Advisor\PolicyAdvisor.xml
Go
Links
Done
My Computer

Policy Advisor for WSE 3.0

Policy Advisor is a security tool for Web Services Enhancements 3.0 for Microsoft .NET (WSE). It examines the configuration and policy files for one or more WSE endpoints, highlights typical security risks, and provides some remedial advice. Moreover, it summarizes the associated trace files (when present), showing message flows between endpoints.

Policy Advisor can be used for reviewing the security of WSE installations. It generates a report that describes errors such as weak or apparently inconsistent security properties, highlights settings that are useful during test but inappropriate in production, and raises some questions to be determined during security reviews. The absence of warnings does not in itself provide strong security guarantees; nonetheless, Policy Advisor can find issues that may otherwise be missed.

This version of Policy Advisor is specific to WSE 3.0; it has the following new features:

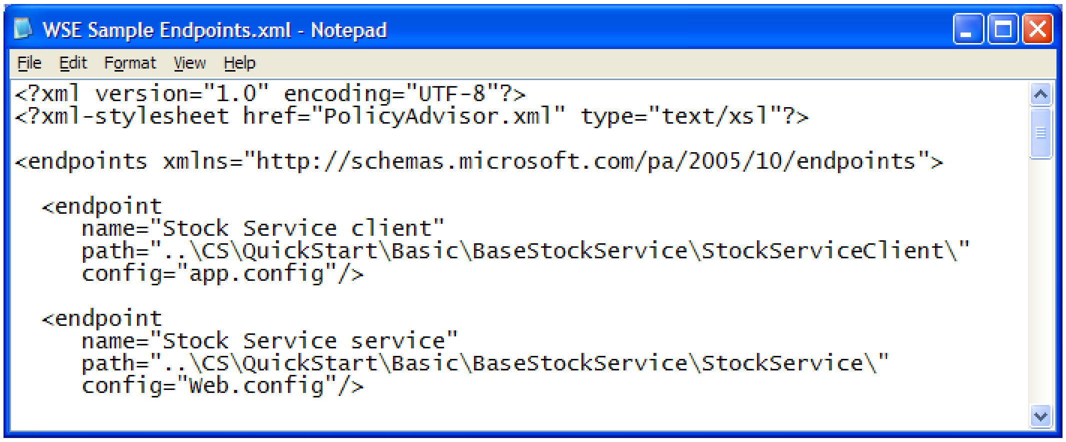
- It supports the new Turnkey Security Assertions.
- It summarizes message flows between multiple endpoints.
- It analyzes multiple configuration, policy, and trace files, as listed in an endpoint file.
- It groups advice by query (instead of policy) to provide a compact report even if many policies are analyzed at once.
- It needs no installation; **PolicyAdvisor.xml** is simply an XSL stylesheet for viewing endpoint files.

Figure A.2
PolicyAdvisor.xml viewed in Internet Explorer

Input Format

An example endpoints file named **WSE Sample Endpoints.xml** is located in the same folder as the **PolicyAdvisor.xml** file. This file lists a selection of the client and server endpoints in the WSE 3.0 samples.

If you open the **WSE Sample Endpoints.xml** file in Notepad, you can see the XML input format, which is a sequence of **endpoint** elements within a root **endpoints** element, as shown in Figure A.3.


WSE Sample Endpoints.xml - Notepad
File Edit Format View Help

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="PolicyAdvisor.xml" type="text/xsl"?>

<endpoints xmlns="http://schemas.microsoft.com/pa/2005/10/endpoints">

  <endpoint
     name="Stock Service client"
     path="..\CS\QuickStart\Basic\BaseStockService\StockServiceClient\"
     config="app.config"/>

  <endpoint
     name="Stock Service service"
     path="..\CS\QuickStart\Basic\BaseStockService\StockService\"
     config="Web.config"/>
```

Figure A.3
***WSE Sample Endpoints.xml** displaying how endpoints are configured in the policy advisor*

An endpoint element may have the following attributes, each of which is optional:

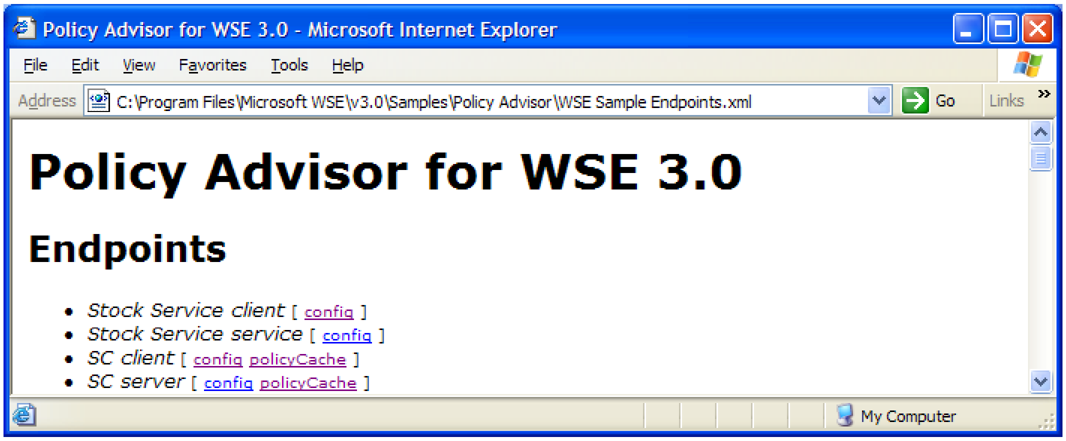
- **name**: This is a name to identify the endpoint in the report that the Policy Advisor generates.
- **path**: This is a base path for the following attributes.
- **config**: This is the configuration file for the endpoint. The concatenation of **path** and **config** is the path to the configuration file.
- **policyCache**: This is the policy file for the endpoint. The concatenation of **path** and **policyCache** is the path to the policy file.
- **input**: This is an existing trace of input messages for the endpoint that when present illustrates its message flow. The concatenation of **path** and **input** is the path to the trace file.
- **output**: This is an existing trace of output messages for the endpoint that when present illustrates its message flow. The concatenation of **path** and **output** is the path to the trace file.

The relative paths are resolved with respect to the folder containing the **PolicyAdvisor.xml** file, not the folder containing the endpoints file, (which is incorrectly stated in the Policy Advisor documentation). This file format is specific to the Policy Advisor tool and contains the XSLT expressions that generate the evaluation report. No other WSE 3.0 components use it.

Use caution when editing the input files. If any of the paths cannot resolve to a file, the XSL engine will fail when running the code in the **PolicyAdvisor.xml** file, which generates an error message, such as: "The system cannot locate the resource specified."

Output Format

If you open the **WSE Sample Endpoints.xml** file in Internet Explorer, you can see a sample report, as shown in Figure A.4.


Policy Advisor for WSE 3.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
Links
My Computer

C:\Program Files\Microsoft WSE\v3.0\Samples\Policy Advisor\WSE Sample Endpoints.xml

Policy Advisor for WSE 3.0

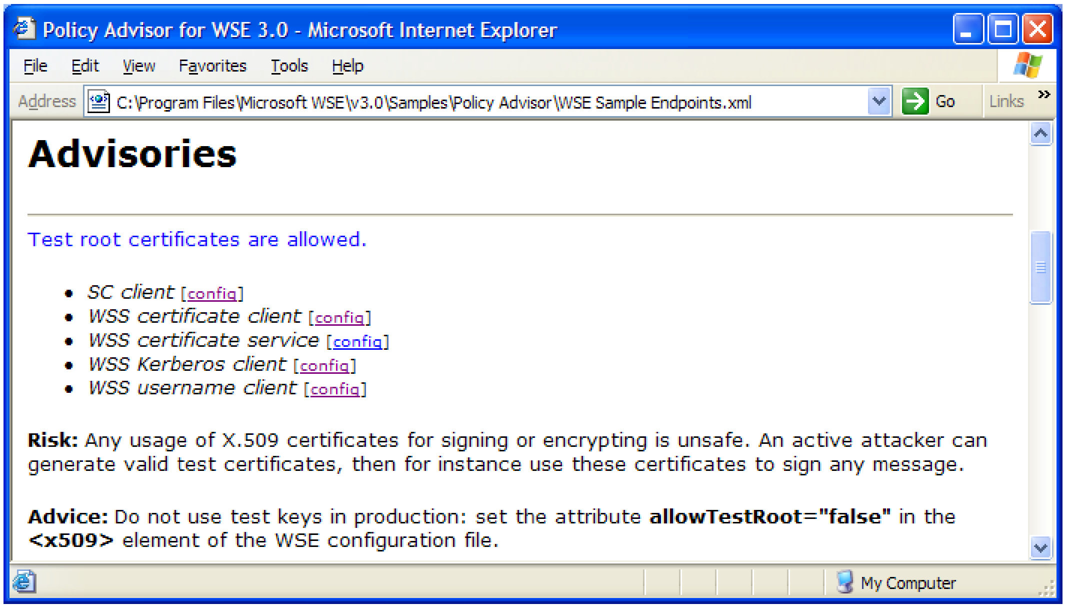
Endpoints

- Stock Service client [config]
- Stock Service service [config]
- SC client [config policyCache]
- SC server [config policyCache]

Figure A.4
***WSE Sample Endpoints.xml** illustrates a Policy Advisor sample report*

The first part of the report lists the names of the endpoints in the input file, and links to the associated files, such as the configuration and policy files.

The next part of the report, shown in Figure A.5, aggregates the results of running a collection of security queries on all the configuration and policy files provided as input. For each query that is triggered, the report includes a one-line summary, a list of the endpoints that triggered the query, a description of the risk, and advice for a suggested action.


Policy Advisor for WSE 3.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address C:\Program Files\Microsoft WSE\v3.0\Samples\Policy Advisor\WSE Sample Endpoints.xml
Go
Links
My Computer

Advisories

Test root certificates are allowed.

- *SC client* [config]
- *WSS certificate client* [config]
- *WSS certificate service* [config]
- *WSS Kerberos client* [config]
- *WSS username client* [config]

Risk: Any usage of X.509 certificates for signing or encrypting is unsafe. An active attacker can generate valid test certificates, then for instance use these certificates to sign any message.

Advice: Do not use test keys in production: set the attribute **allowTestRoot="false"** in the **<x509>** element of the WSE configuration file.

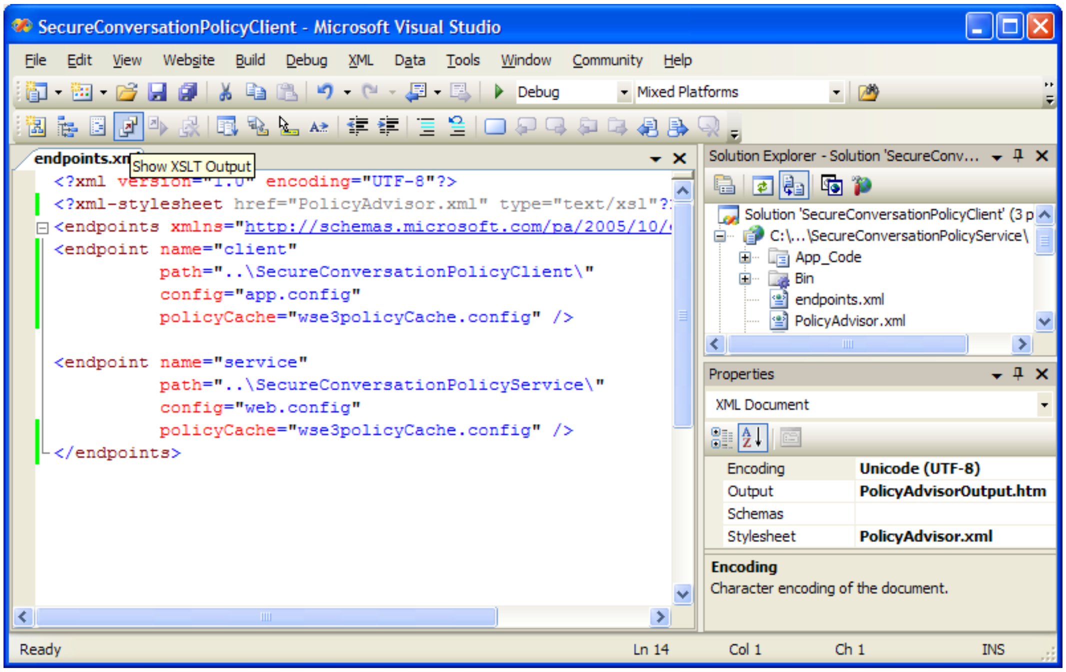
Figure A.5
WSE Sample ***Endpoints.xml*** *illustrating how the Policy Advisor tool issues advisories*

The report describes issues such as weak or apparently inconsistent security properties, shows settings that are useful during test, but inappropriate in production, and raises some questions that you can address during security reviews.

As well as presentational markup, the Extensible Hypertext Markup Language (XHTML) output includes **<instance>** elements that contain the raw results of queries. This means that it is possible to use batch scripts to run the Policy Advisor tool and then extract the raw data of the report to compare it with previous reports.

Using Policy Advisor with Visual Studio 2005

You can include an endpoints file in a project and invoke Policy Advisor directly in Visual Studio 2005, as shown in Figure A.6.


SecureConversationPolicyClient - Microsoft Visual Studio
File Edit View Website Build Debug XML Data Tools Window Community Help
Debug
Mixed Platforms
endpoints.xml
Show XSLT Output
Solution Explorer - Solution 'SecureConv...
Solution 'SecureConversationPolicyClient' (3 p
C:\...\SecureConversationPolicyService\
App_Code
Bin
endpoints.xml
PolicyAdvisor.xml
Properties
XML Document
Ready
Ln 14
Col 1
Ch 1
INS

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="PolicyAdvisor.xml" type="text/xsl"?
<endpoints xmlns="http://schemas.microsoft.com/pa/2005/10/
<endpoint name="client"
          path="..\SecureConversationPolicyClient\"
          config="app.config"
          policyCache="wse3policyCache.config" />

<endpoint name="service"
          path="..\SecureConversationPolicyService\"
          config="web.config"
          policyCache="wse3policyCache.config" />
</endpoints>
```

Encoding	Unicode (UTF-8)
Output	PolicyAdvisorOutput.htm
Schemas	
Stylesheet	PolicyAdvisor.xml

Encoding
Character encoding of the document.

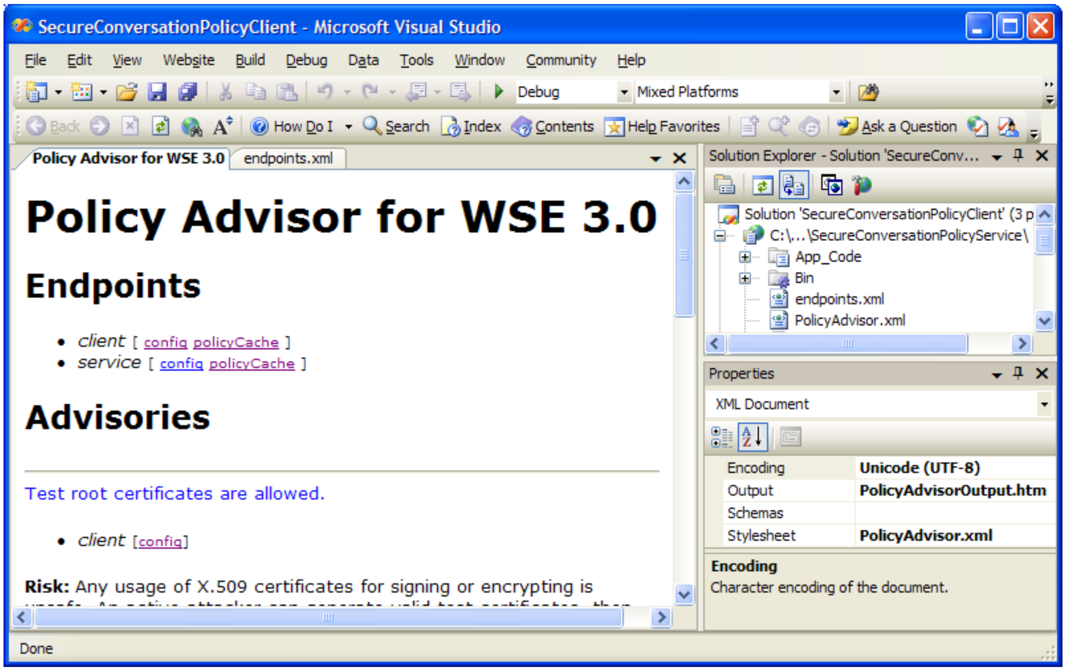
Figure A.6
Using Policy Advisor in Visual Studio 2005

To invoke Policy Advisor in Visual Studio 2005, perform the following steps.

► **To use Policy Advisor in Visual Studio 2005**

1. Open the solution, then in the Solution Explorer, right-click the project and click **Add Existing Item**.
2. Navigate to the directory in WSE 3.0 where the policy advisor sample is installed. By default, it is located at C:\Program Files\Microsoft WSE\v3.0\Samples \Policy Advisor.
3. Select the **PolicyAdvisor.xml** file and click **Add**.
4. In the Solution Explorer, right-click the project and click **Add New Item**.
5. In the same directory as the **PolicyAdvisor.xml** file, locate the **WSE Sample Endpoints.xml** file, select it, and then click **Add.**
6. In the Solution Explorer, right-click the **WSE Sample Endpoints.xml** file, select **Rename**, and then rename the file as **endpoints.xml**.
7. In the Solution Explorer, double-click the **endpoints.xml** file to open it.

8. Identify as many **<endpoint>** elements in the file for as many applications as you want to run against the Policy Advisor tool.
9. Delete the remaining **<endpoint>** elements from the file.
10. Update the **name** attribute of each **<endpoint>** element that remains in the file with the name that you want to use for the endpoint.
11. Update the **path** attribute of each **<endpoint>** element to point to the project folder for that endpoint. Ensure that a back slash "\" appears at the end of the path.
12. Update the **config** attribute of each **<endpoint>** element to point to the configuration file for that application. This is usually "App.config" or "Web.config" for client applications and Web applications, respectively.
13. Update the **policyCache** attribute of each **<endpoint>** element to point to the policy cache file for that application. If you used the default settings to configure policy on the application, the policy cache file name is "wse3policyCache.config."
14. In the **Properties** window, specify the output location in the **Output** property. This is usually an .htm file, such as "PolicyOutput.htm."
15. Specify the Stylesheet as the **PolicyAdvisor.xml** file that you added in Step 3.
16. On the toolbar, click the **Show XSLT Output** button to display the results of the policy analysis of your configured applications as shown in Figure A.7.


SecureConversationPolicyClient - Microsoft Visual Studio
File Edit View Website Build Debug Data Tools Window Community Help
Debug
Mixed Platforms
Back How Do I Search Index Contents Help Favorites Ask a Question
Policy Advisor for WSE 3.0
endpoints.xml
Solution Explorer - Solution 'SecureConv...
Solution 'SecureConversationPolicyClient' (3 p
C:\...\SecureConversationPolicyService\
App_Code
Bin
endpoints.xml
PolicyAdvisor.xml
Properties
XML Document
Done

Policy Advisor for WSE 3.0

Endpoints

- client [config policyCache]
- service [config policyCache]

Advisories

Test root certificates are allowed.

- client [config]

Risk: Any usage of X.509 certificates for signing or encrypting is

Encoding	Unicode (UTF-8)
Output	PolicyAdvisorOutput.htm
Schemas	
Stylesheet	PolicyAdvisor.xml

Encoding
Character encoding of the document.

Figure A.7
The Policy Advisor output file that displays in Visual Studio 2005

Patterns: A Common Vocabulary for Information Technology Professionals

Overview

In the last decade or so, Microsoft and its competitors have focused a lot of attention on finding better ways to capture, persist, and organize information gained through analysis of data, so that knowledge can be shared effectively between people.

Early knowledge management solutions focused on capturing knowledge in documents and providing access through rudimentary tagging and search mechanisms. Over time, more sophisticated solutions have evolved that incorporate advanced collaboration technologies alongside sophisticated tools that generate taxonomies for the knowledge in an organization.

This paper demonstrates the need for the Information Technology industry to focus on establishing a similar knowledge management solution to increase the effectiveness of communications among software engineers including architects, software designers, developers, and testers. Such a solution will increase the effectiveness of application development, and will also increase our ability to communicate clearly and consistently within large organizations and across organizational boundaries.

The Challenge

Communication across large organizations is difficult. Some key factors are organizational, cultural, time and geographical. For software engineers in particular, the problem has increased over time due to the lack of a single, standard mechanism for persisting knowledge about proven software designs which has often resulted in, at best, inconsistent reinvention of the wheel, and at worst, overlapping or duplicated functionality.

It is interesting to note that these symptoms are similar to those facing knowledge workers in organizations without sophisticated knowledge management solutions. Workers duplicate efforts, quality is compromised through inconsistent analysis, and the organization is less able to leverage agility to its competitive advantage.

Due to the similarity of these symptoms our hypothesis is that a similar solution to the one that is currently being deployed for knowledge workers is required for architects and developers.

The Solution

The solution is a knowledge management solution for software engineers. Such a solution should incorporate the following capabilities:

- A standard notation for describing proven architectural designs that incorporates information such as when the design is applicable, tradeoffs associated with using the design and a solution that is reproducible.
- A standard vocabulary to describe architectural designs.
- A searchable repository for publishing, sharing, and locating architectural designs.
- A layered model for describing taxonomies of architectural designs, including support for composite designs.
- A means of incorporating architectural designs, and models based upon those designs, into integrated development environments such as Visual Studio.

In the last few years, the IT industry has thought about many of these capabilities and has developed solutions with various degrees of acceptance. The Microsoft patterns & practices team is also working on a set of solutions to each of these challenges with the goal being to contribute to the development and adoption of a common vocabulary across the IT industry.

We will briefly describe individual solutions that are available with the goal being to encourage and increase adoption of patterns as a standard means of communicating architectural designs.

A Standard Notation for Designs — Design Patterns

Patterns provide an effective means of communicating best practices for solving recurring design challenges. Patterns use a template that incorporates a pattern name, the context in which the pattern exists, a description of the problem the pattern solves, a solution to the problem, and consequences or tradeoffs that arise from using the pattern.

In the IT industry we often remark how similar a problem is on latest technologies to something we solved a decade ago on technologies long since retired. A good example of this is the way in which we used to design CICS applications on a mainframe — an approach called pseudo-conversational development was a primary design pattern for building scalable CICS applications. This varied from the conversational (and significantly less scalable) equivalent used within the TSO environment on a mainframe.

The conceptual lessons learned from the mainframe are just as applicable to development in .NET and J2EE. In fact, had our industry standardized on a standard vocabulary for describing such problems 20 years ago it might be simpler to transition developers from the mainframe to newer technologies. At minimum the issues surrounding how state and connections should be maintained in scalable online application would have been better understood and resulted in a lot fewer eCommerce organizations experiencing outages due to poor resource management when their applications experienced spikes in traffic.

A significant benefit from using design patterns is their inherent longevity — in many cases architecture and design patterns outlast the platform upon which they were first described. The Microsoft patterns & practices team deliberately separates the implementation of patterns from the corresponding architectural or design patterns that they implement. This allows the implementation to be demonstrated on multiple products (for example, .NET Enterprise Services, WCF, and BizTalk) and allows the implementation to be replaced as our technologies evolve (for example, from WSE 2.0 to WSE 3.0 to WCF to…).

It is difficult to measure the success of a design pattern or even patterns-based guidance. Consider the impact that the original *Design Patterns: Elements of Reusable Object-Oriented Software* book has had on the software industry, were it measured in terms of actual sales it would probably be considered moderately successful — but would never appear on a New York Times best seller list. Now, consider how frequently you hear software engineers, often working in different teams, organizations, or geographic locations, describe solutions to problems by referencing patterns such as the Façade or Abstract Factory — and you get some insight into the value of patterns and their importance as part of the software engineering vocabulary.

A Standard Vocabulary — Pattern Languages

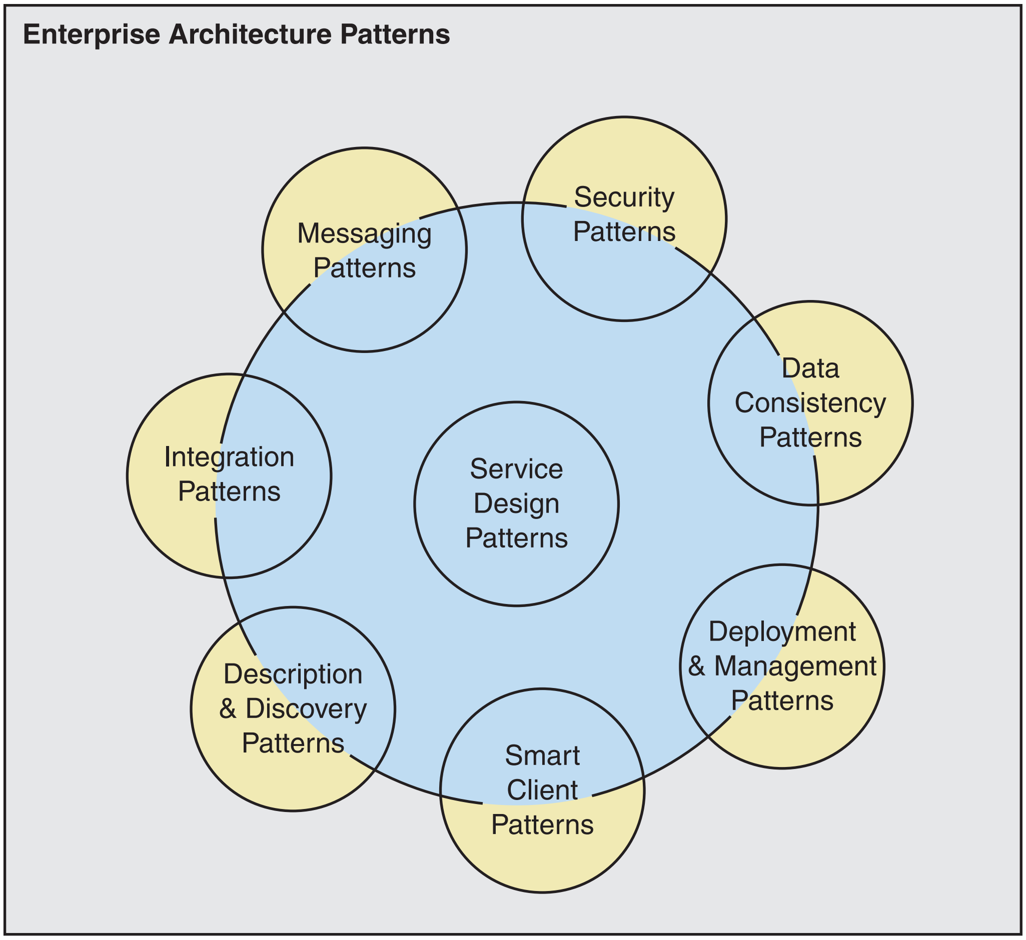
Design patterns are named so that the solutions they encompass can be communicated and discussed with clarity. Over time, the names establish a common understanding of the key characteristics of their implementation, just as has been the case for algorithms such as Bubble Sort and Quick Sort.

There is an emerging trend for design patterns to be created as a group to help establish vocabulary within a particular domain. Gregor Hohpe's *Enterprise Integration Patterns*[1] is a good example — it provides a common vocabulary for architects focusing on integration and messaging.

People often reminisce about similarities between technologies that are prevalent for developing distributed applications. This may or may not be true, but imagine how much simpler it would be to move developers from J2EE, CORBA, and COM+ to .NET and SOA if all distributed applications had a common understanding of key message exchange patterns and an understanding of the roles and responsibilities of key concepts, such as the naming service, the stub, skeleton, and even IDL. Sure, new messaging patterns would emerge as our technologies evolve, but these patterns would build off a well established base reducing the learning curve and expediting adoption of new technologies.

[1] Hohpe, Gregor, and Bobby Woolf, *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*, Reading, MA: Addison-Wesley Professional, 2003.

The Microsoft patterns & practices team has been working on a series of architecture and design patterns focused on the domain of service orientation. The first release establishes a vocabulary around Web service security, with future releases aimed to focus on messaging and data consistency. If used effectively, such a vocabulary allows underlying products to change, while still helping developers to understand core concepts that need to be addressed by each pattern.


Enterprise Architecture Patterns
Messaging Patterns
Security Patterns
Data Consistency Patterns
Integration Patterns
Service Design Patterns
Deployment & Management Patterns
Description & Discovery Patterns
Smart Client Patterns

🟦 Patterns related to Service Orientation

🟨 Patterns not directly related to Service Orientation

Figure A.8
Patterns of Service Orientation

A Standard Repository — PatternShare

Any knowledge management solution requires a means of publishing and discussing content. The Microsoft patterns & practices team has developed a content management solution called PatternShare that is based on a wiki. PatternShare allows developers across the world to publish patterns that they use. PatternShare encourages other developers to apply, discuss and even edit the patterns that are posted.

PatternShare also provides access to the patterns using full-text search or through the use of a visual model based on a layered view of a particular domain. The existing search capabilities must evolve overtime to allow people to find patterns based on problems they are trying to solve. Search results displayed within visual information models should help software engineers locate patterns more easily. Some ideas for this visualization are proposed in the next section. For more information, see PatternShare. PatternShare is an open wiki and you are invited to participate in its growth.

A Layered Model — The Pattern Frame

As described earlier, the ability to describe architectural and design level challenges independent of their implementation allows the patterns to truly transcend the life of a particular product. A great example of this is provided by the original *Design Patterns: Elements of Reusable Object-Oriented Software* design patterns — they initially provided implementations in C++, but all of these patterns have since been implemented in C, SmallTalk, Java, C#, and Visual Basic. NET.

The organizational frame used within Microsoft's Enterprise Solution patterns provides a basic model that allows patterns to be organized not just by levels of abstraction — but also based on the area of technology with which the pattern is focused.

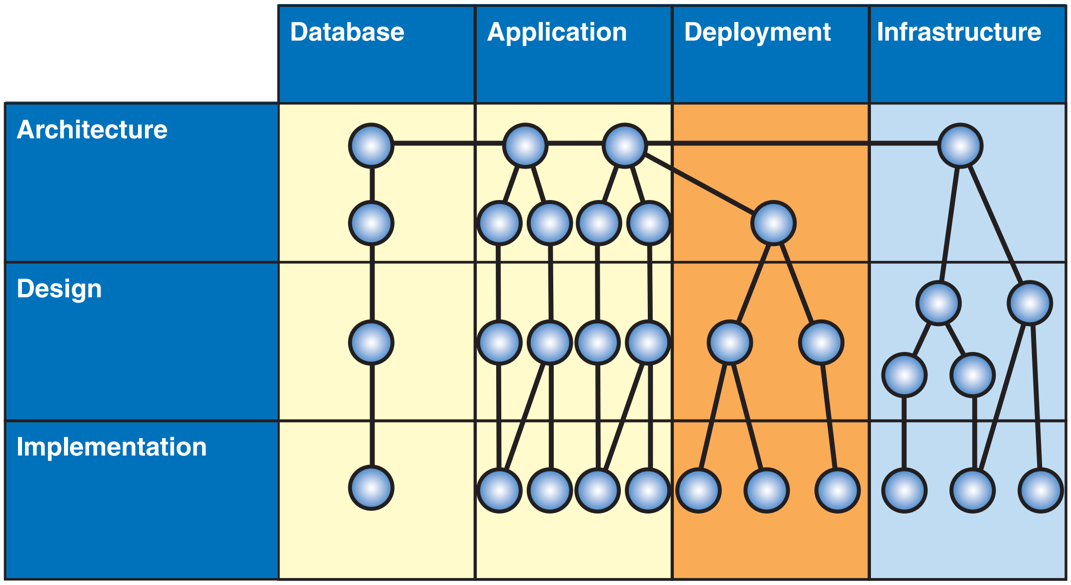


Figure A.9
The basic model provided by Microsoft's Enterprise Solution patterns

Frequently, guidance to software engineers on the use of technologies is at the implementation level — which means that every time a technology changes, the guidance needs to change and software engineers have to relearn how to solve the same problem with new technologies. Focusing on architectural and design level guidance, and then showing implementations on particular products, is not only more efficient — it also establishes a group memory for software engineers.

A more sophisticated model has also been developed that allows organizations to describe enterprise architectures in terms of patterns — allowing for increased visibility into the organization's architecture and underlying relationships between systems. For more information on this model, see Describing the Enterprise Architectural Space.

More importantly, PatternShare incorporates a visual model that provides an example of how a series of patterns can be organized. This helps developers searching for guidance on a particular problem to find what patterns exist. Please see Enterprise Architectural Space Organizing Table for more on this example.

IDE Integration — Guidance Automation Toolkit (GAT)

A critical aspect of empowering developers in using patterns as the basis for describing designs is to incorporate such patterns into IDE's such as Visual Studio. As mentioned earlier, this is important for enterprise developers who are often more capable of addressing complex business problems (such as how to calculate housing loan interest rates) than they are at solving complex technical challenges (such as ensuring a password stored within a database is stored using an appropriate hashing mechanism).

The Guidance Automation toolkit, developed by patterns & practices, is an extension to Visual Studio 2005 that allows architects to author rich, integrated user experiences for reusable assets including frameworks, components and patterns.

A pattern such as Direct Authentication, which simply describes how a client and a service can authenticate using a shared secret, can appear trivial at the surface. However, when you consider aspects such as: how to ensure the shared secret is stored in a database in a secure (hashed) format, how to hash the shared secret, whether to hash it on the client or the service and what the implications are for securing the message — you get some idea of why a pattern describing what the associated best practices are, is valuable.

If you then examine the number of factors that you need to consider when implementing such a simple solution — for example, a custom **UsernameToken** manager, a custom hashing algorithm, WSE 3.0 security policy and modifications to the Web.config file and associated Web service interface — it becomes clear why patterns and their implementations in Visual Studio are so important.

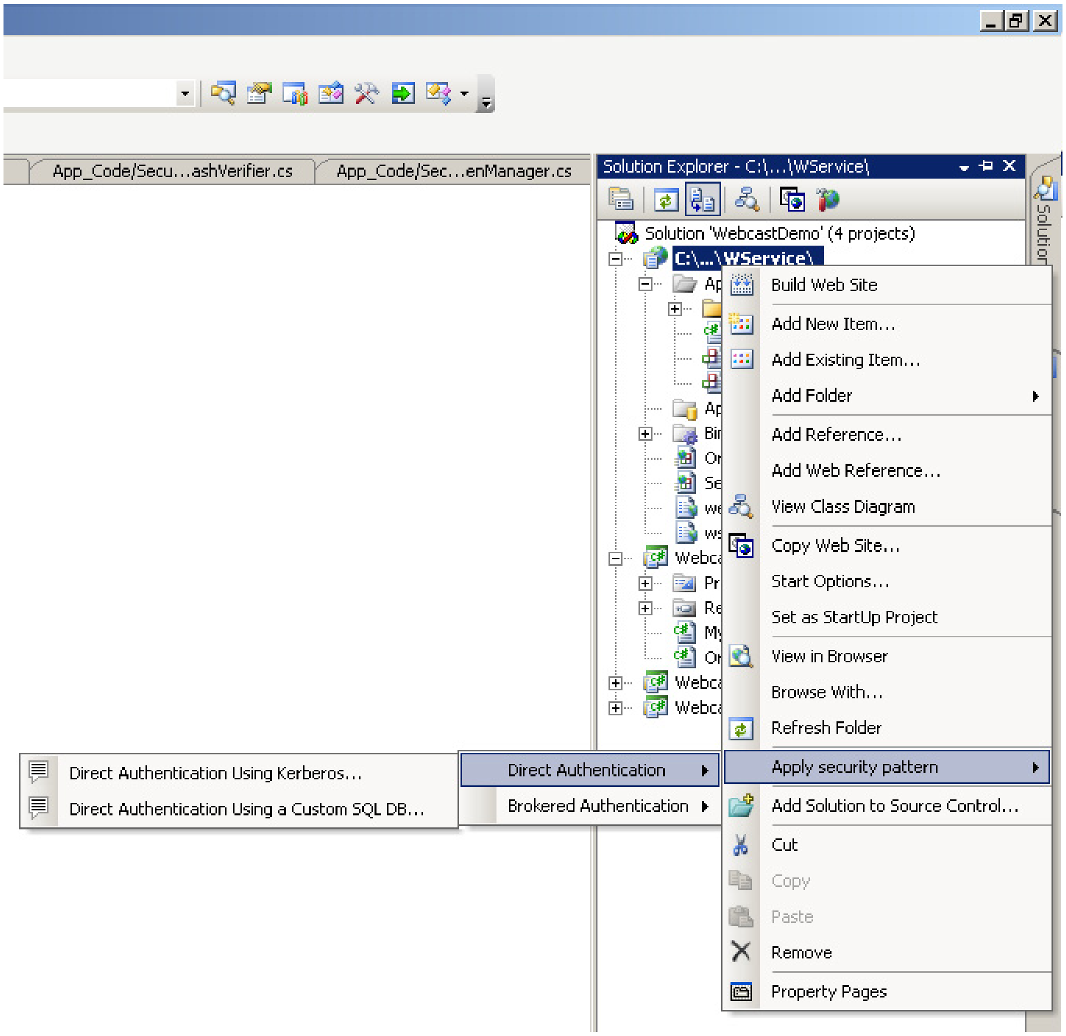


Figure A.10

A prototype of GAT based guidance

Figure A.10 is a prototype of GAT based guidance that has been developed to accompany the patterns & practices Web service security pattern's initiative. For more information, join the Web Service Security community workspace.

Conclusion and Recommendation

A pattern based knowledge management solution will allow software engineers to start capturing proven software designs and reusing them across organizations. The pattern language that naturally emerges from such a solution will also, over time, form a technical vocabulary that will increase effectiveness of communications — not just across teams within an organization but also across our industry — resulting in better communication, increased productivity and of course better quality.

The capabilities presented within this paper already exist in varying degrees of maturity. In some cases the capabilities have existed for at least a decade, and in others the capabilities exist only as prototype work within the Microsoft patterns & practices team. To truly reap the benefits of a unified knowledge management solution for software engineers, all of these capabilities must be considered collectively.

Author: Jason Hogg, Microsoft Corporation

Reviewer and Editor: Paul Slater, Wadeware LLC

Reviewers: Per Vonge Nielsen, Edward Jezierski, David Trowbridge, Matt Deacon, Microsoft Corporation; Ward Cunningham

More Information

Hohpe, Gregor, and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Reading, MA: Addison-Wesley Professional, 2003, ISBN: 0321200683.

Enterprise Solution Patterns Using Microsoft .NET, Redmond: Microsoft Press, 2003, ISBN: 0735618399. Also available on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/Esp.asp*.

Web Service Security: Scenarios, Patterns, and Implementation Guidance community workspace: *http://go.microsoft.com/fwlink/?linkid=52393&clcid=0x409*.

Describing the Enterprise Architectural Space: *http://msdn.microsoft.com/practices/guidetype/Guides/default.aspx?pull=/library/en-us/dnpag/html/entarch.asp*.

PatternShare: *http://www.patternshare.org/*.

Enterprise Architectural Space Organizing Table: *http://patternshare.org/default.aspx/Home.EnterpriseArchitecturalSpaceOrganizingTable*.

Gamma, Eric, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley Professional, 1995, ISBN: 0201633612.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Hoboken, NJ: John Wiley & Sons, 1996, ISBN: 0471958697.

Glossary

This section contains a brief summary of key terms and definitions that appear in *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements 3.0*. This glossary is not intended to be an authoritative or comprehensive security glossary for this guide because many such resources already exist. The numbers at the end of the definition indicate where terms are directly cited from one of the resources in the "References" section.

authentication
The process of identifying an individual using the credentials of that individual. For example, a bank teller may be required to authenticate who you are by examining your driver's license. Authentication typically occurs immediately after identification.

authorization
The process of determining whether an authenticated subject is allowed to access a resource or perform a task within a security domain. Authorization uses information about a client's identity and/or roles to determine the resources or tasks that a client can perform.

Brokered authentication
A type of authentication where a trusted authority is used to broker authentication services between a client and a service. An example is shown in Figure A.11.

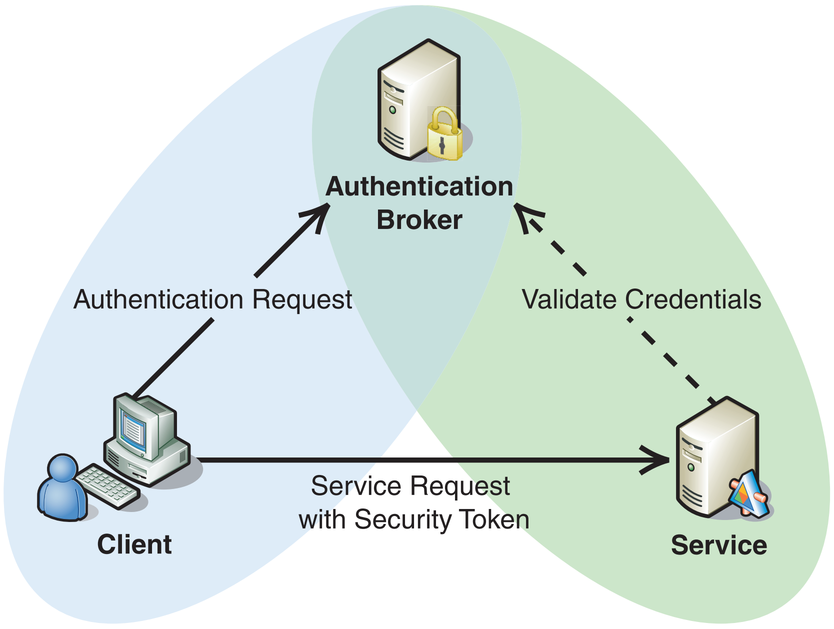


Figure A.11
Using a broker to perform authentication when a client and service do not share a trust relationship

claim
A claim is a declaration made by an entity. Examples include name, identity, key, group, privilege, and capability. [2]

client
The client accesses the Web service. The client provides credentials for authentication during the request to the Web service.

confidentiality
A process by which data is protected so that only authorized actors or security token owners can view the data.

credentials
A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential.

data confidentiality
The encrypting of message data so that unauthorized entities cannot view the contents of the message.

data integrity
The verification that a message has not changed in transit.

Data origin authentication
Data origin authentication takes data integrity a step further by supporting the ability to identify and validate the origin of a message.

data encryption
Encryption is the process of converting data (plaintext) into something that appears to be random and meaningless (ciphertext), which is difficult to decode without a secret key. Encryption is used to provide message confidentiality.

delegation
A process where the service account is allowed to access a remote resource on behalf of another Windows account, which is typically the client accessing a service.

digital signature
This is an asymmetric signature that is created with the private key of a client. Digital signatures can be used to support non-repudiation requirements.

Direct authentication
A type of authentication where the service validates credentials directly with an identity store, such as a database or directory service. When both the client and service participate in a trust relationship that allows them to exchange and validate credentials including passwords, direct authentication can be performed, as shown in Figure A.12.

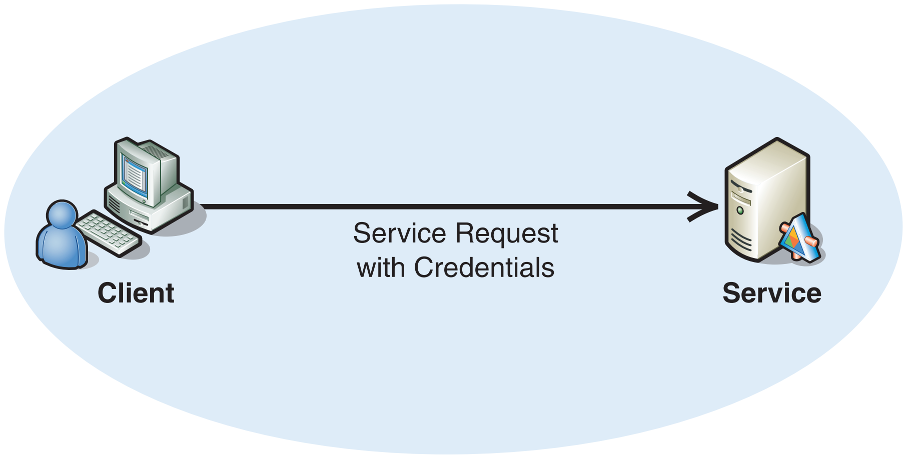


Figure A.12
Direct authentication when a client and service share a trust relationship

identification
Represents the use of an identifier that allows a system to recognize a particular subject and distinguish it from other users of the system.

impersonation
Impersonation is the act of assuming a different identity on a temporary basis so that a different security context or set of credentials can be used to access a resource.

impersonation/delegation model
A resource access model that flows the security context of the original caller through successive application tiers and onto back-end resource managers. This allows resource managers to implement authorization decisions based on the identity of the original caller. This is in contrast to the trusted subsystem model. [1]

message layer security
Message layer security represents an approach where all the information that is related to security is encapsulated in the message. In other words, with message layer security, the credentials are passed in the message.

mutual authentication
Mutual authentication is a form of authentication where the client authenticates the server in addition to the server that authenticates the client. [1]

proof-of-possession
A value that a client presents to demonstrate knowledge of either a shared secret or a private key to support client authentication. Proof-of-possession that uses a shared secret can be established using the actual shared secret, such as a user's password, or a password equivalent, such as a digest of the shared secret, which is typically created with a hash of the shared secret and a salt value. Proof-of-possession can also be established using the XML signature within a SOAP message where the XML signature is generated symmetrically based on the shared secret or asymmetrically based on the sender's private key.

protection scope
This term describes the scope of protection for a Web service message. Protection scope refers to the extent the message will be protected, whether it is for its entire message lifetime or only while it is in transit between servers.

protocol transition
Protocol transition is a process where the service account transitions an identity that was authenticated using a non-Windows protocol into a Windows security context.

public-private key encryption
Public-private key encryption is an asymmetric form of encryption that relies on a cryptographically generated public/private key pair. Data encrypted with a private key can only be decrypted with the corresponding public key (and vice-versa).

security context
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 an access token combine to form the security context of the process. [1]

security context token (SCT)
A lightweight token that can be established for multiple message exchanges between two endpoints using the protocol defined in the WS-SecureConversation specification. [4]

security token
A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential. Most security tokens will also contain additional information that is specific to the authentication broker that issued the token.

security token service (STS)
A Web service that issues security tokens (see WS-Security). An STS makes assertions based on evidence that it trusts, to whomever trusts it (or to specific recipients). To communicate trust, a service requires proof, such as a signature to prove knowledge of a security token or set of security tokens. An STS can generate tokens or it can rely on a separate STS to issue a security token with its own trust statement. (Note that for some security token formats, this can be nothing more than a re-issuance or co-signature). This process forms the basis of trust brokering. [3]

service account
This is the Windows account that the operating system process uses when it hosts a service. Web services are usually hosted in a process managed by an application server, such as Internet Information Services (IIS) that performs operations using the identity of a service account.

signed security token
A signed security token is a security token that is asserted and cryptographically signed by a specific authority, such as an X.409 certificate or a Kerberos ticket. [2]

service
A service is a Web service that requires authentication.

transport layer security
Transport layer security represents an approach where security protection is enforced by lower level network communication protocols.

trust
Trust is the characteristic that one entity is willing to rely upon a second entity to execute a set of actions and/or to make a set of assertions about a set of subjects and/or scopes. [2]

trusted subsystem
This is a process where a trusted business identity is used to access a resource on behalf of the client. The identity could belong to a service account or it could be the identity of an application account created specifically for access to remote resources.

References

For more security glossary information, see the following resources:

1. "Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication" on MSDN: *http://msdn.microsoft.com/practices/Topics/security/default.aspx?pull=/library/en-us/dnnetsec/html/SecNetAPgl.asp*.
2. "Web Services Security: SOAP Message Security 1.0 (WS-Security 2003)" on the Oasis Web site: *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.
3. "Web Services Trust Language (WS-Trust)" on MSDN: *http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-trust.pdf*.
4. "Managing Security Context Tokens in a Web Farm" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/sctinfarm.asp*.

Bibliography

This section contains a consolidated list of referenced resources that appear in the *Web Service Security* guide.

General Information

The following references provide useful background information that will help you gain a better overall understanding of this guide.

Security Background

Brown, K. *The .NET Developer's Guide to Windows Security*, Reading, MA: Addison-Wesley Professional, 2005, ISBN: 0321228359.

Kaufman, C., Perlman, R., and Speciner, M. *Network Security — PRIVATE Communication in a PUBLIC World*. Upper Saddle River, NJ: Prentice Hall PTR., 2002, ISBN: 0130460192.

Improving Web Application Security: Threats and Countermeasures on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp*.

Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch10.asp*.

"Threat Modeling Web Applications" on MSDN: *http://msdn.microsoft.com/practices/Topics/security/default.aspx?pull=/library/en-us/dnpag2/html/tmwa.asp*.

Security Challenges, Threats and Countermeasures Version 1.0 on the WS-I Web site: *http://www.ws-i.org/Profiles/BasicSecurity/SecurityChallenges-1.0.pdf*.

OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/*.

Pattern Resources

PatternShare: *http://www.patternshare.org/*.

Gamma, Eric, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley Professional, 1995, ISBN: 0201633612.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Hoboken, NJ: John Wiley & Sons, 1996, ISBN: 0471958697.

Hohpe, Gregor, and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Reading, MA: Addison-Wesley Professional, 2003, ISBN: 0321200683. Also available on: *http://www.eaipatterns.com*.

Enterprise Solution Patterns Using Microsoft .NET, Redmond: Microsoft Press, 2003, ISBN: 0735618399. Also available on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/Esp.asp*.

Integration Patterns, Redmond: Microsoft Press, 2004, ISBN: 073561850X. Also available on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/intpatt.asp*.

Chapter 1, "Authentication Patterns"

For more information about authorization on the .NET Framework, see "Authentication and Authorization" in *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* on MSDN: *http://msdn.microsoft.com/practices/Topics/security/default.aspx?pull=/library/en-us/dnnetsec/html/SecNetch03.asp*.

For more information about the Kerberos protocol specifications, see RFC 1510: The Kerberos Network Authentication Service (V5): *http://www.faqs.org/rfcs/rfc1510.html*.

For more information about Kerberos authentication in Windows Server 2003, see "Kerberos Authentication Technical Reference" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/b748fb3f-dbf0-4b01-9b22-be14a8b4ae10.mspx*.

For a general overview of PKI technologies, see "PKI Technologies" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/6d5d9ef3-75ca-46c1-acf6-57dc7e9a6adf.mspx*.

For more information about WS-Trust, see *Web Services Trust Language (WS-Trust)* on MSDN: *http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-trust.pdf*.

For more information about ADFS, see "Introduction to ADFS" on Microsoft TechNet: *http://technet2.microsoft.com/WindowsServer/en/Library/c67c9b41-1017-420d-a50e-092696f40c171033.mspx*.

For more information about Security Assertion Markup Language (SAML), go to the OASIS Web site: *http://www.oasis-open.org/specs/index.php#samlv1.1*.

For more information about WS-SecureConversation, see *Web Services Secure Conversation Language (WS-SecureConversation)* on MSDN: *http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-secureconversation.pdf*.

For more information about SAML 1.1 core specification, go to the Oasis Web site: *http://www.oasis-open.org/specs/index.php#samlv1.1*.

For more information about SAML token profile 1.0, see *Web Security Services: SAML Token Profile* on the Oasis Web site: *http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf*.

Chapter 2, "Message Protection Patterns"

For more information on WS-Security version 1.0, see the OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/specs/index.php#wssv1.0*.

For more information about HMAC, see RFC 2104 — HMAC: Keyed Hashing for Message Authentication: *http://www.ietf.org/rfc/rfc2104.txt?number=2104*.

Chapter 3, "Implementing Transport and Message Layer Security"

For information about Web Services Security, see "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)": *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.

For information about derived key tokens, see "Web Services Secure Conversation Language (WS-SecureConversation)": *http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf*.

For information about how to configure a **SqlMembershipProvider**, see "How To: Use Membership in ASP.NET 2.0" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000022.asp*.

For information about creating a custom ASP.NET 2.0 membership provider, see "Building Custom Providers for ASP.NET 2.0 Membership" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/bucupro.asp*.

For information about configuring WSE 3.0 to prevent replay attacks, see "Web Services Enhancements 3.0 <replayDetection> Element" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/b4fa188d-4804-40bd-877b-c01058555013.asp*.

For more information about performance objectives, see "Improving .NET Performance and Scalability" on MSDN: *http://msdn.microsoft.com/practices/Topics/perfscale/default.aspx?pull=/library/en-us/dnpag/html/scalenet.asp*.

For information about WSE 3.0 policy, see "Securing a Web Service" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/7b8f29da-22d5-4e03-b645-15011a80e548.asp*.

For information about Kerberos assertion policy settings, see "<kerberosSecurity> Element" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/bde6a6dd-00e4-4c37-aa8d-8821f2f25bc5.asp*.

For more information about performance objectives see, "Improving .NET Performance and Scalability" on MSDN: *http://msdn.microsoft.com/practices/Topics/perfscale/default.aspx?pull=/library/en-us/dnpag/html/scalenet.asp*.

For information about installing X.509 certificates in the local certificate store, see "How to: Use the X.509 Certificate Management Tools" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse/html/21eb7fb5-bd11-4cce-be0c-7b3d0cd14acb.asp?frame=true*.

For information about how to install X.509 certificates in the local machine certificate store, see "Certificates How To" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/ServerHelp/fb037b9f-8956-411c-a3e8-ce1dfe37da11.mspx*.

For more information on configuring the behavior of X.509 security in WSE 3.0, see "<x509> Element" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/72b7b9c9-63dd-4ce7-a25f-e40b164912d2.asp* in the WSE documentation.

For information about how to set the **findType** and **findValue** attributes for the **<x509>** element, see "<x509> Element (Policy)" in the WSE 3.0 documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/4caad727-778e-4c57-90f8-0edca69eed1f.asp*.

For information about configuring other settings for this policy assertion, see "<mutualCertificate10> Element" in the WSE 3.0 documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/973d38d8-6347-4617-983f-089e64a2b02c.asp*.

To learn more about Windows Integrated Security, see the "Authentication and Authorization Strategies" section in "Web Services Security" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch10.asp*.

To call a Web service configured to use Windows Integrated Authentication, see the "Specifying Client Credentials for Windows Authentication" section in "Web Services Security" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch10.asp*.

To learn how to configure IIS for HTTP basic authentication, see "Basic Authentication in IIS 6.0" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/abbca505-6f63-4267-aac1-1ea89d861eb4.mspx*.

To learn how an SSL session is established between two parties, see "Description of the Secure Sockets Layer (SSL) Handshake" on Microsoft Help and Support: *http://support.microsoft.com/default.aspx?scid=kb;%5bLN%5d;Q257591*.

To learn about how a client authenticating to a service using SSL operates, see "Description of the Client Authentication Process During the SSL Handshake" on Microsoft Help and Support: *http://support.microsoft.com/kb/257586/EN-US/*.

To learn how to implement SSL, see:

- "How To Set Up SSL on a Web Server" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod30.asp*.
- "How To Call a Web Service Using SSL" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod28.asp*.
- "How To Call a Web Service Using Client Certificates from ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod27.asp*.

To learn how to call a Web service that requires credentials, see the "Passing Credentials for Authentication to Web Services" section in "Web Services Security" on MSDN: *http://msdn.microsoft.com/library/en-us/secmod/html/secmod10.asp*.

For more information about implementing transport layer security with Kerberos and IPSec on Windows Server 2003, see "IPSec" on Microsoft.com: *http://www.microsoft.com/windowsserver2003/technologies/networking/ipsec/default.mspx*.

For more information about XML performance guidance in the .NET Framework, see Chapter 9, "Improving XML Performance," in *Improving .NET Application Performance and Scalability* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt09.asp*.

Chapter 4, "Resource Access Patterns"

For more information about Web services security, see "Web Services Security" in *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch10.asp*.

For more information about using impersonation and delegation in ASP.NET 2.0, see "How To: Use Impersonation and Delegation in ASP.NET 2.0" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000023.asp*.

For more information about designing the authentication and authorization mechanisms for a distributed ASP.NET Web application, see "Authentication and Authorization" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod03.asp*.

For more information about developing identity-aware applications, see "Developing Identity-Aware ASP.NET Applications, Identity and Access Management Services" on MSDN: *http://www.microsoft.com/technet/security/topics/identitymanagement/idmanage/P3ASPD_1.mspx*.

Chapter 5, "Service Boundary Protection Patterns"

For more information about idempotent methods, see "9 Method Definitions": *http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*.

For more information about idempotent, see "Idempotent" on the Wikipedia Web site: *http://en.wikipedia.org/wiki/Idempotent*.

For more information about idempotent Web services, see "Idempotent Receiver" on the Enterprise Integration Patterns Web site: *http://www.eaipatterns.com/IdempotentReceiver.html*.

For more information about SOAP Message Security, see OASIS: "Web Services Security: SOAP Message Security 1.0 (WS Security 2004)": *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.

For more information about SQL Server performance optimization, see "Optimizing Database Performance Overview" on MSDN: *http://msdn.microsoft.com/library/?url=/library/en-us/optimsql/odp_tunovw_9mxz.asp*.

For more information about security best practices for SQL Server 2000, see "SQL Server 2000 SP3 Security Features and Best Practices" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/sp3sec00.mspx*.

Chapter 4, "Design Guidelines for Secure Web Applications," in *Improving Web Application Security: Threats and Countermeasures* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html /THCMCh04.asp*.

For more information about **<httpRuntime>**, see "**<httpRuntime>** Element" in the *.NET Framework General Reference* on MSDN: *http://msdn.microsoft.com/library /default.asp?url=/library/en-us/cpgenref/html/gngrfhttpruntimesection.asp*.

For more information about WSE 3.0 policy assertions, see "Policy Assertions" on MSDN: *http://msdn.microsoft.com/library/?url=/library/en-us/wse3.0/html/1d3257fd-fcfb -45cf-beca-3cfcefceaa8b.asp*.

For more information about using the **SoapClient/SoapService** classes for messaging, see "How To: Send and Receive a SOAP Message by Using the **SoapClient** and **SoapService** Classes," in the WSE 3.0 documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/8cbdb522 -0672-4c17-b68e-0d3e65067271.asp*.

For more information about adding a schema to a resource file see "Resolving the Unknown: Building Custom XmlResolvers in the .NET Framework," on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnxmlnet/html /CusXmlRes.asp*.

For more information about implementing regular expressions, see "How To: Use Regular Expressions to Constrain Input in ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000001.asp*.

For more information about using regular expressions in XML Schemas, see "XML Schema Regular Expressions" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/html/ea72d044-6b46-4124-b6dc-95976e411b4a.asp*.

For more information about XML performance guidance in the .NET Framework, see Chapter 9, "Improving XML Performance," in *Improving .NET Application Performance and Scalability* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt09.asp*.

For more information about how to create the event source that the Web service uses, see the "Creating a New Event Source at Install Time" section of "How To: Use the Network Service Account to Access Resources in ASP.NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000015.asp*.

For more information about creating custom Policy Assertions in WSE 3.0, see "Custom Policy Assertions" in the WSE 3.0 product documentation on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/5636c932-30d0-42c6-ac17-88c40b5935b8.asp*.

Chapter 6, "Service Deployment Patterns"

"Service Interface Pattern" in *Enterprise Solution Patterns Using Microsoft .NET* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesServiceInterface.asp*.

For more information about using the WseWsdl3.exe utility, see the "WSDL to Proxy Class Tool" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/fbefe453-3851-439b-9c10-fb036b59ff81.asp*.

For more information on referral cache syntax, see "How to: Configure the WSE SOAP Router" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/6414f229-cead-48af-a293-cb893c24c0e6.asp*.

For more information about implementing SOAP routers in WSE 3.0, see: "Routing SOAP Messages with WSE" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/b41230fb-d0e1-48b1-88c0-3daf7a40c9e8.asp*.

For more information about XML performance guidance in the .NET Framework, see Chapter 9, "Improving XML Performance," in *Improving .NET Application Performance and Scalability* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt09.asp*.

Chapter 7, "Technical Supplements"

For information about compatibility issues between GSSAPI and the Kerberos SSP, see "SSPI/Kerberos Interoperability with GSSAPI" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthn/security/sspi_kerberos_interoperability_with_gssapi.asp*.

For information about replay detection with the sequence field, see section "5.3.2 Authenticators" in RFC 1510: *http://www.ietf.org/rfc/rfc1510.txt*.

For in-depth troubleshooting information for the Kerberos protocol implementation in Windows 2000 and Windows 2003, see "Troubleshooting Kerberos Delegation" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/tkerbdel.mspx*.

For information about Kerberos authentication, see "What Is Kerberos Authentication?" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/792ed95d-6f13-4181-a218-e4eaab361c1b.mspx*.

For information about certificate policies, see "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework": *http://www.ietf.org/rfc/rfc2527.txt*.

For information about X.509 PKI services on Windows Server 2003, see "Designing a Public Key Infrastructure" on Microsoft TechNet: *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/DepKit/b1ee9920-d7ef-4ce5-b63c-3661c72e0f0b.mspx*.

For information about the MakeCert utility, see "Certificate Creation Tool (Makecert.exe)" on MSDN: *http://winfx.msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_fxtools/html/b0343f8e-9c41-4852-a85c-f8a0c408cf0d.asp*.

For information about PKI and Windows Server 2003, see "Public Key Infrastructure for Windows Server 2003": *http://www.microsoft.com/windowsserver2003/technologies/pki/default.mspx*.

For information about the Online Certificate Status Protocol (OCSP), see "RFC 2650, X.509 Internet Public Key Infrastructure Online Certificate Status Protocol — OCSP": *http://www.ietf.org/rfc/rfc2560.txt*.

For information about the Certificate Services PKI solution in Windows Server 2003, see "What Is Certificate Services?": *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/63e3ba1c-cc23-40b1-9ca2-853869677318.mspx*.

For more information about certificates, see "What are certificates?" on the RSA Laboratories Web site: *http://www.rsasecurity.com/rsalabs/node.asp?id=2277*.

For information about Secure Sockets Layer (SSL), see "What is SSL?" on the RSA Laboratories' Web site: *http://www.rsasecurity.com/rsalabs/node.asp?id=2293*.

For more information about WS-Security version 1.0, see the OASIS Standards and Other Approved Work (including WS-Security) on the OASIS Web site: *http://www.oasis-open.org/specs/index.php#wssv1.0*.

For information about IPSec, see "Internet Protocol Security (IPsec) Operations Topics": *http://www.microsoft.com/technet/prodtechnol/windowsserver2003/operations/ipsec.mspx*

For information about the Internet X.509 PKI certificate and CRL profile, see "Internet X.509 Public Key Infrastructure Certificate and CRL Profile" (RFC 2459): *http://www.ietf.org/rfc/rfc2459.txt*.

Appendix

For in-depth information about interoperability and Web service security, see *WS-I Basic Security Profile 1.0 Reference Implementation: Preview release for the .NET Framework version 1.1* on MSDN: *http://msdn.microsoft.com/practices/guidetype/RefImp/default.aspx?pull=/library/en-us/dnpag2/html/MSWSIBSP.asp*.

For information about the implemented specifications, see the "WSE 3.0 documentation": *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse3.0/html/d0ed7f06-504b-40f8-939c-b884ffce77c0.asp*.

For information about the WS-I Basic Security Profile, see "Basic Security Profile Version 1.0": *http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html*.

For information about SOAP Message Security 1.0, see "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) from OASIS": *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.

Describing the Enterprise Architectural Space: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/entarch.asp*.

For more security glossary information, see the following resources:

- "Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication" on MSDN at *http://msdn.microsoft.com/practices/Topics/security/default.aspx?pull=/library/en-us/dnnetsec/html/SecNetAPgl.asp*.
- "Web Services Security: SOAP Message Security 1.0 (WS-Security 2003)" on the Oasis Web site at *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*.
- "Web Services Trust Language (WS-Trust)" on MSDN at *http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-trust.pdf*
- "Managing Security Context Tokens in a Web Farm" on MSDN at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/sctinfarm.asp*.

Community Workspace and Wiki

To post questions, provide feedback, or connect with other users for sharing ideas, visit the community workspace “Web Service Security: Scenarios, Patterns, and Implementation Guidance”: *http://go.microsoft.com/fwlink/?LinkId=57044*.

To add new problem/solution links related to this guidance, see the “Web Service Security Wiki”: *http://go.microsoft.com/fwlink/?LinkId=57051*.