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.

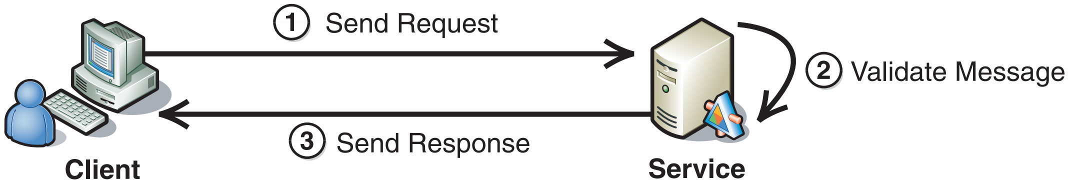

Figure 5.4
Message validation occurring at a Web service

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.

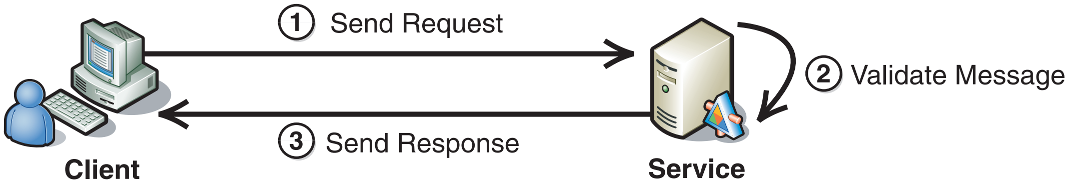

Figure 5.5
Validating a request message

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.