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.

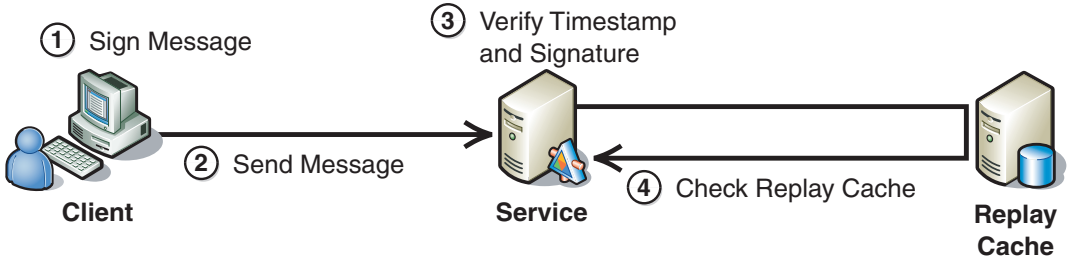


**Figure 5.2**

*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.

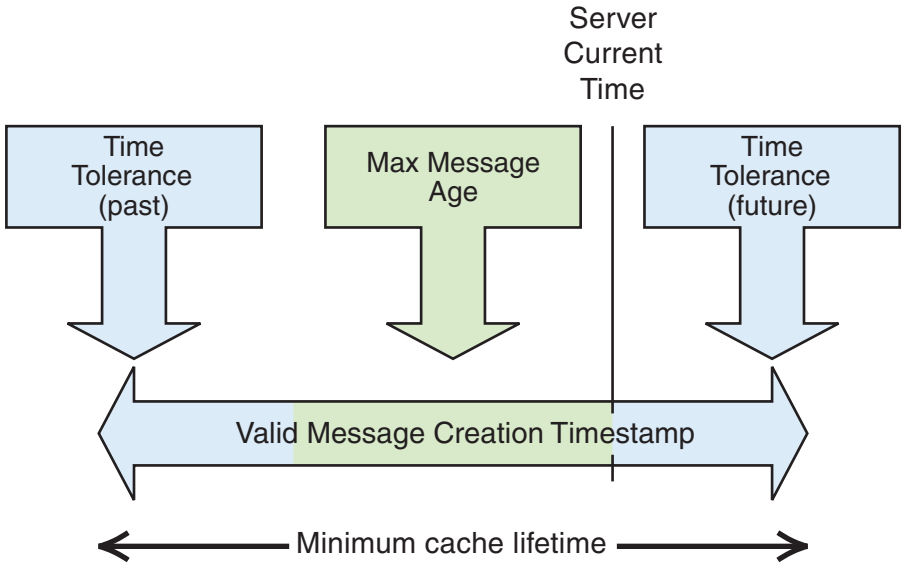


**Figure 5.3**
*The relationship between the configuration settings*

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?