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.

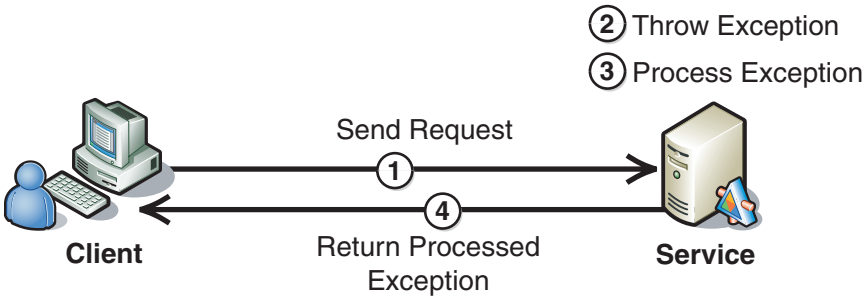

Figure 5.7

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*.