**Summary**

This two-part article series provides an in-depth analysis of the interoperability issues regarding SAP’s and Microsoft’s Web Services Security (WS-Security) implementation. Based on a real-world B2B scenario from the retail industry, the first part assesses the security risks and recommends the appropriate security configuration from the various options provided by WS-Security. The second part explains the configuration settings and shows code samples to secure the communication based on the requirements identified in the first article. It also considers the requirements from the Basic Security Profile (BSP) 1.0 Working Group Draft released by the Web Services Interoperability (WS-I) Organization and looks at the interoperability issues when using the declarative security features in WSE.

**By:** Martin Raepple

**Company:** SAP AG

**Date:** September 16, 2005

**Table of Contents**

- Architecture of the VMI application ...................................................................................... 2
- Walkthrough .......................................................................................................................... 2
- Application Design .............................................................................................................. 3
- Roll up your sleeves .............................................................................................................. 4
- Project Import .................................................................................................................... 4
- Keys and Certificates .......................................................................................................... 5
- Securing the Product Activity Notification ........................................................................... 8
- Configure WSE to accept UsernameTokens without nonce and creation time fields ........12
- Securing the Purchase Order Message ............................................................................... 12
- Securing Purchase Order Confirmation ............................................................................. 14
- Declarative security policy enforcement ............................................................................. 17
- Conclusion .......................................................................................................................... 19
Part II

Part 1 [1] introduced the VMI scenario, analyzed the security risk regarding the business process and derived a security architecture based on Web Services Security (WS-Security). Building on this foundation, this article will describe a step-by-step guide to safeguard the wholesaler inventory system based on J2EE and the SAP NetWeaver Web Application Server (Web AS) 6.40 and the supplier replenishment application based on the Microsoft .NET Framework. It is assumed that you have a basic understanding of object oriented programming, J2EE and the C# programming language.

Architecture of the VMI application

Before we dive deep into the code and security configuration of the VMI scenario [2], we will have a closer look at the architecture of the application. Let’s start with a walkthrough of the VMI application introduced in the first part of this article, followed by a more technical description of its components.

Walkthrough

The following diagram illustrates the core sequence of application actions triggered by the user interface and the Web Service calls.

![Diagram of VMI application architecture](image-url)
1. Via the wholesaler’s inventory, sales and purchase order processing application, customers (retailers) can login using a Web Browser interface and place sales orders from the list of items in the wholesaler’s inventory. Each sales order triggers a call of InventoryManager Web Service’s orderItem operation that updates the stock level accordingly.

2. For simplicity, sales data is not collected by the wholesaler and send on a scheduled basis to the supplier. Instead, each inventory update immediately generates a Product Activity message for the supplier by calling the submitProductActivity operation of its Replenishment Web Service.

3. If the new stock levels of the ordered items fall below the mutually agreed threshold, the supplier calculates the number of items needed by the wholesaler (based on a very complex algorithm) to replenish the stock and sends a (yet unconfirmed) purchase order (PO) by calling the submitPurchaseOrder operation of the PurchaseOrderManager Web Service.

4. Users owning the role of inventory managers can check the current inventory levels and view the list of purchase orders via a Web-based UI.

5. By confirming a new purchase order, the UI calls to the confirmPurchaseOrder operation of the wholesaler’s PurchaseOrderManager Web Service which in turn changes the status to ‘confirmed’.

6. This also triggers a call of the Replenishment Service’s confirmPurchaseOrder operation in order to synchronize the status with the supplier and start the production and delivery process.

Application Design

The application design on both sides basically follows best practices in terms of programming patterns and technologies according to the respective platform.

The wholesaler’s J2EE-based inventory and order management application is subdivided into the three common layers presentation, business logic and persistency. The presentation layer utilizes Web Dynpro for building the user interface (UI) and thus follows closely the Model-View-Controller pattern. User authentication and authorization is handled by the User Management Engine (UME) which provides a rich framework to configure role-based access management. The core business logic for managing the inventory and purchase orders resides in two session enterprise java beans (EJB, PurchaseOrderManagerBean and InventoryManagerBean), which both define a coarse-grained, service-oriented interface according to the J2EE session façade pattern [3]. Any database access is directed by the session beans using the local interfaces of container managed persistence (CMP) entity EJBs for the inventory (InventoryBean) and purchase order (PurchaseOrderBean and PurchaseOrderItemBean) database tables. In order to provide the model (i.e. business logic) to the UI, the session beans are exposed as Web Services (PurchaseOrderManagerService and InventoryManagerService). In addition, a restricted view on the PurchaseOrderManagerBean is published as a Web Service (PurchaseOrderManager) to the outside world based on the concept of Virtual Interfaces, exposing only the submitPurchaseOrder operation.

The supplier’s part of the VMI application consists of only two components: The ReplenishmentService is an ASP .NET Web Service offering operations to submit the Product Activity messages.
Web Services Security Interoperability with SAP NetWeaver and Microsoft .NET (Part II)

(submitProductActivity) and confirm POs (confirmPurchaseOrder). A simple Windows Forms-based UI for the supplier is used to track the status of the POs sent to the wholesaler. Interprocess communication between the Web Service and the UI is implemented using .NET remoting.

Roll up your sleeves

Now it’s time to get your hands dirty and start implementing the security mechanism for the VMI sample application. To install and run the application successfully, please make sure that your development environment fulfills the following prerequisites:

- SAP NetWeaver Web AS Java 6.40 SP 13
- SAP J2EE Engine Visual Administrator with access to the J2EE engine
- SAP NetWeaver Developer Studio (NWDS)
- Microsoft Visual Studio .NET 2003 (includes .NET Framework SDK 1.1)
- Microsoft Web Service Enhancements (WSE) 2.0 Service Pack 3 [4]

Project Import

For detailed instructions on how to deploy the wholesaler and supplier application parts with NWDS and Visual Studio please take a look at the README.TXT file in the ZIP-archive [1] that contains the code for the complete VMI application. After successful import, you should have the following projects in your NWDS workspace:

- HolidayEJB: Core business logic and Web Services of the wholesaler’s application
- HolidayUI: Web Dynpro project for the wholesaler’s Web user interface
- HolidayDataDictionary: Data dictionary for the inventory and purchase order database tables
- WDAction: Local Development Component that contains the deployable UME permissions archive file (actions.xml)
- HolidayEAR: Enterprise Archive Project for the wholesaler application
- LotionIncClientProxy: Deployable client proxy classes for the supplier’s Replenishment Service.

1 It is recommended to choose the Visual Studio .NET option when installing WSE since this will embed the new configuration tools that come with WSE right into Visual Studio.
The assemblies for the supplier's application solution have the following outline in Visual Studio:

- **ReplenishmentManager**: Windows Forms application to track the purchase order confirmation status
- **ReplenishmentService**: ASP .NET Web Service for the Replenishment Service using the WSE extensions

### Keys and Certificates

All interactions using XML Signature and XML Encryption require the use of asymmetric key pairs, consisting of a private key (to sign or decrypt messages) and a public key (to encrypt messages or verify a signature), wrapped in X.509 certificate. For testing purposes we will use self-signed certificates and won't cover the practice of issuing certificate requests to a trusted Certification Authority (CA) as it would be required in a production environment. Nevertheless, we will follow good security practice and use different key pairs for signing and encrypting messages. The following diagram illustrates where the keys are stored and used to secure the messages between the wholesaler and supplier.
The wholesaler uses the Key Storage Service within the Web AS to generate and store the key pairs for signing Purchase Order confirmation (HolidaySig) and decrypting the Product Activity message (HolidayEnc). The certificates of both key pairs must also be imported into the supplier’s key store so that the wholesaler’s signature can be verified and confidential PO data can be encrypted by the supplier. The setup procedure is as follows:

- Start the J2EE Engine Visual Administrator and login to the running engine.
- Choose the node <SID>-Server → Services → Key Storage (where <SID> is the system ID of the J2EE engine) and activate the tabs Runtime (top) and Data (bottom) in right window.
- In the area View, select the key store view ‘WebServiceSecurity’.
- In the Entry section (bottom right), choose Create, and generate a new signature key pair with the Common Name (CN) value ‘Holiday Corp.’, entry name ‘Signature’, algorithm ‘RSA’ and key length 1024. Don’t forget to activate the checkbox ‘Store Certificate’. Click on ‘Generate’. This key will be used to sign the purchase order confirmations.
- In the Entries list for the WebServiceSecurity key store view, select the public key certificate for the new signature key (‘Signature-cert’) and export it for later import by the supplier as a X.509 certificate file (e.g. HolidaySig.crt) in a temporary directory.
Once again, generate a new encryption key pair with the CN value ‘Holiday Corp. Encryption’, entry name ‘XMLEncryption’, algorithm ‘RSA’ and key length 1024 (also check ‘Store Certificate’). This key will be used to encrypt the purchase order messages. Export the new encryption key (‘XMLEncryption-cert’) as a X.509 certificate file (e.g. HolidayEnc.crt) in a temporary directory.

Import the public key certificates into the supplier’s trusted certificates key store. For each file, locate it in the temporary directory, choose ‘Install certificate’ from the context menu and use the ‘Certificate Import Wizard’ to import it in the supplier’s ‘Trusted Root Certificate Authorities’ key store.

Although the Key Storage Service in Visual Administrator also allows generating signature key pairs based on the Digital Signature Algorithm (DSA), it is recommended to use RSA for .NET interoperability since neither Microsoft’s X.509 Token Manager supports DSA nor is it a valid option from the perspective of the WS-I Basic Security Profile (BSP).

The supplier requires only one key pair (LotionEnc) to encrypt the Product Activity messages sent by the wholesaler. For the .NET framework, the Certificate Creation tool makecert.exe generates public and private key pairs and stores it in a certificate file. When issuing the command

```
makecert.exe -n "CN=Lotion Inc. Encryption" -sk LotionEnc -ss my -sr LocalMachine -r –sky exchange LotionEnc.cer
```

a self-signed certificate file (LotionEnc.cer) is generated (using the option –r) and the private key to decrypt messages is stored in the supplier’s personal key store (using the option –ss my). It is important to include –sky exchange as the certificate will be later used for encryption.

The new private key is not trustworthy and cannot be used to decrypt messages unless its root CA certificate (in the case of a self signed certificate the same certificate) is added to the supplier’s Trusted Root Certificate Authorities store on the same server where the Web service is installed using the Certificate Import Wizard via the context menu of the LotionEnc.cer file.

In addition, the Replenishment Web Service must have permission to access the private key file. Use the WSE Certificate Tool to access the file properties of the private key and allow the ‘Users’ group permission to access the key:

- Start the WSE Certificate Tool (WseCertificate2.exe) from the WSE Tools\Certificates subdirectory (e.g. C:\Program Files\Microsoft WSE\v2.0\Tools\Certificates)
- Choose the certificate with the name ‘Lotion Inc. Encryption’ and click on the ‘View Private Key File Properties’ button.
- In the file properties dialog, activate the tab ‘Security’, add the local ‘Users’ group for testing purposes (see screenshot below) and confirm the settings.
Finally, import the certificate file of the supplier’s encryption key into the J2EE engine’s key storage with Visual Administrator:

- Go to the Key Storage service and select the key store view ‘WebServiceSecurity’
- In the Entry section (bottom right), choose ‘Load’, and select the certificate file `LotionEnc.cer` created by the `makecert` tool.

Now that all required keys and certificates are created and stored in their corresponding key stores, we can configure the Web Services to secure the message flow.

**Securing the Product Activity Notification**

The countermeasures for the Product Activity Notification were determined as follows:

- Tampering and Eavesdropping: Encrypted message body using the supplier’s encryption key (LotionEnc)
- Spoofing: UsernameToken, encrypted using the supplier’s encryption key (LotionEnc)
- Message Replay: Timestamp created by the wholesaler

The required steps to configure the wholesaler application using NWDS are:

- Switch to the Web Services Perspective (Windows → Open Perspective → Web Services)
- Open the `ReplenishmentServiceSoap` logical port from the `net.lotioninc.wsproxy.inventory.ProductActivityProxy` client proxy and activate the Security tab
- Configure ‘Document Authentication’ and ‘Basic (username/password)’ for the authentication mechanism. From the list of operations, select `submitProductActivity` and configure ‘Encryption+Username’ as the request policy template.
Save the changes, rebuild the EAR file and deploy it to the J2EE engine.

After deployment, the security runtime parameters of the proxy client can be configured with Visual Administrator:

- First, please make sure that the proxy (and any other secured Web Service component deployed on the J2EE engine) has the proper access rights to read from the WebServiceSecurity key store view. Go to Services -> Security Provider, activate the Runtime/Policy Configurations tab and choose the "keystore-view.WebServiceSecurity" entry from the list of components. Switch to the ‘Security Roles’ tab, select the ‘view-creator’ from the ‘Security Roles’ section and check if the group ‘Everyone’ is listed in the ‘Mappings’ section.

- Select the Web Services Security service, activate the Runtime/Profile Administration tab and choose ‘Outbound messages’ to create a new profile with the name ‘EncryptedUsernameToken’ as illustrated below.

- Next, assign the predefined policy template ‘Encryption+Username’ to the new outbound profile and enter a username and password of a valid Windows user account registered with the supplier. WSE 2.0 automatically authenticates incoming UsernameTokens (via LogonUser API call for local user authentication) when the password is supplied in plain text. It is also possible to write a custom UsernameToken manager for WSE (e.g. to authenticate credentials stored in a database), but this is out of the scope of this article. Further configure the key store view (WebServiceSecurity) and the key that will be used for encryption, in this case the supplier’s public encryption key certificate (LotionEnc).
Having the outbound profile configured according to the security policy, it needs to be assigned to the `submitProductActivity` operation by configuring it in the Document Security of the `net.lotioninc.wsproxy.inventory.ProductActivityProxy` (Web Services Security / Runtime → Security Configuration → Web Service Clients → sap.com → LotionIncClientProxy).

To process the encrypted request message with the UsernameToken correctly at the supplier, some additional configuration and code is required:

- Although the supplier’s Replenishment service is able to decrypt the message using the properly installed private key without further coding, we still have to make sure from a security policy standpoint that the message actually is encrypted and contains a UsernameToken for origin authentication. The listing below shows the code required at the beginning of the `submitProductActivity` operation before any processing logic takes place. It basically gets the message context from the current request and loops through the collection of security tokens and encrypted message parts found in the request. The code will throw a SOAP fault back to sender if any other element than the body of the message has been encrypted and not exactly one username token has been found.

```csharp
// ***********************************************
// check security policy of incoming request
// ***********************************************
SoapContext reqContext = RequestSoapContext.Current;

// check for SOAP Request
if (reqContext == null)
    throw new ApplicationException("Not a valid SOAP request");
// check for Username Token
```
UsernameToken unToken = null;
{
    if (st is UsernameToken)
    {
        if (unToken != null)
        {
            throw new SoapException("Found more than one Username Token", SoapException.ClientFaultCode);
        }
        else
        {
            unToken = (UsernameToken)st;
        }
    }
}
if (unToken == null)
    throw new SoapException("No Username Token found in request", SoapException.ClientFaultCode);

// check for encrypted body
bool validEncryption = false;
{
    if (objElem is EncryptedData)
    {
        // Encrypted Data exists in the Element collections.
        EncryptedData encData = objElem as EncryptedData;
        // check if it is the body that was encrypted
        if (encData.TargetElement.LocalName == "Body")
        {
            validEncryption = true;
        }
    }
}
if (!validEncryption)
    throw new SoapException("Body not encrypted in request", SoapException.ClientFaultCode);

- Sending the Product Activity notification without further configuration of the supplier’s Replenishment service results in the Java SOAPFaultException, because the default configuration of WSE assumes that any incoming UsernameToken will always contain a nonce and creation time field.
Web Services Security Interoperability with SAP NetWeaver and Microsoft .NET (Part II)

protect replay of the token. Thus the replay cache does not function and throws a WSE System.Exception:
System.Exception: WSE567: The incoming Username token must contain both a nonce and a creation time for the replay detection feature.

Configure WSE to accept UsernameTokens without nonce and creation time fields

In order to let WSE accept the UsernameToken provided in the request messages sent by the wholesaler, one needs to disable the replay cache in WSE by adding the following lines to the (WSE) <security> section in the Web.config XML configuration file of the Replenishment Service:

<security>
  ...
  <securityTokenManager xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" qname="wsse:UsernameToken">
    <replayDetection enabled="false" />
  </securityTokenManager>
  ...

Securing the Purchase Order Message

After receipt of the sales and stock data update the supplier decides whether the wholesaler needs to replenish its inventory. If so, it will send a Purchase Order protected by the following security measures:

- Tampering and Eavesdropping: Encrypted message body using the wholesaler's encryption key (HolidayEnc)
- Spoofing: UsernameToken, encrypted using the wholesaler’s public encryption key (HolidayEnc)
- Message Replay: Timestamp created by the supplier

To send a Purchase Order message according to this policy the required tokens and encryption programmatically must be added to the supplier's Replenishment Service by enhancing the complex procurement calculation logic in the processProductActivity method of the ProductActivityProcessor class.

First we'll instantiate the client proxy of the wholesaler's PurchaseOrderManager Web Service that can be created using Microsoft's .NET tool wsdl.exe and needs to be derived from the WSE class Microsoft.Web.Services2.WebServicesClientProtocol in order to use WS-Security features:

    PurchaseOrderManager serviceProxy = new PurchaseOrderManager();
The UsernameToken added to the request context of the service proxy instance must provide a login name of any valid J2EE user and will send a plain text password. Since this is a testing environment it is acceptable to provide the password in cleartext:

```csharp
// create UsernameToken
UsernameToken unToken;
string username = "LotionInc";
string password = "eruces";
unToken = new UsernameToken(username, password, PasswordOption.SendPlainText);
// add the username token to the request
serviceProxy.RequestSoapContext.Security.Tokens.Add(unToken);
```

To encrypt the UsernameToken and the body of the Purchase Order request, the public key certificate of the wholesaler’s encryption key is loaded from the trusted root store, using its CN name as the unique identifier:

```csharp
// Retrieve the X.509 certificate from the certificate store
X509SecurityToken binToken = null;
X509CertificateStore store = X509CertificateStore.LocalMachineStore(
    X509CertificateStore.RootStore);
store.OpenRead();
X509CertificateCollection certs =
    store.FindCertificateBySubjectString("Holiday Corp.");
// Get the first certificate in the collection
binToken = new X509SecurityToken( ((X509Certificate) certs[0]) );
if (binToken == null)
    throw new ApplicationException("Unable to obtain security token.");
// close the certificate store
store.Close();
```

Finally, the UsernameToken and message body are encrypted using the X.509 token.

```csharp
// Replace the old UsernameToken with the encrypted UsernameToken
    new EncryptedData(binToken, "#" + unToken.Id));

// Encrypt the SOAP message using the X.509 token
```

The submitPurchaseOrder operation of the PurchaseOrderManager Service is configured accordingly with NWDS and Visual Administrator:
In the Web Services Perspective, open the HolidayEJB project from the EJB Explorer

Double click on Web Service Configurations and select the entry PurchaseOrderManager → external Security from the list of available configurations

Configure ‘Document Authentication’ and ‘Basic (username / password)’ as the required authentication mechanism for UsernameToken and choose ‘Encryption+Username’ for the request policy template.

In Visual Administrator, select the Web Services Security service, activate the Runtime/Security Administration tab, and open the Document Security settings for the PurchaseOrderManager Web Service (Security Configuration → Web Services → sap.com → HolidayEAR → PurchaseOrderManager*external)

Assign the predefined profile ‘default Encryption+Username’ to the inbound policy of the submitPurchaseOrder operation.

Securing Purchase Order Confirmation

The purchase order confirmation is different from the first two messages since it does not carry any confidential data but introduces the use of a digital signature. To recap from part one, this last step in the process requires to legally bind the identity of the sender (wholesaler) to the transaction and provide a strong authentication mechanism to the service provider.

Let’s start again with the configuration procedure at the service requestor which are the generated proxy classes used by the wholesaler application to access the Replenishment Service. Since you cannot apply different authentication policies at the operation level of a service proxy, you have to split the configuration into separate proxies. Therefore, two proxies exist in the LotionIncClientProxy project to access the Replenishment Service (ProductActivityProxy and ConfirmPurchaseOrderProxy) from the wholesaler application. Each of them is configured differently according to the security requirements of the corresponding process step. To attach a signature to the Purchase Order confirmation message, follow this procedure:
Switch to the Web Services Perspective in NWDS, open the LotionIncClientProxy project in the Client Explorer and open the ReplenishmentServiceSoap port from the net.lotioninc.wsproxy.purchaseorder.ConfirmPurchaseOrderProxy service proxy.

Activate the ‘Security’ tab and configure ‘Document Authentication’ and ‘X.509 Certificate’ for the authentication mechanism. From the list of operations, select confirmPurchaseOrder and choose ‘Signature’ from the list of available request policy templates.

Save the changes, rebuild the EAR file of the Deployable Proxy project and deploy it to the J2EE engine.

After deployment, the security runtime parameters of the Purchase Order Confirmation proxy client can be configured with Visual Administrator:

Select the Web Services Security service, activate the Runtime/Profile Administration tab and choose ‘Outbound messages’ to create a new profile with the name SignatureHolidayCorp.

As illustrated below, assign the predefined policy template Signature to the new outbound profile and configure the key store view (WebServiceSecurity) and the key (Signature) that will be used to sign any request messages.
Save the changes made to the client proxy runtime settings.

To enforce the security policy at the supplier, the `confirmPurchaseOrder` method programmatically searches the request for a valid signature of the body and the timestamp by looping through the collection of security elements found in the request:

```csharp
SoapContext reqContext = RequestSoapContext.Current;
// check for signed body
bool validSignature = false;

{
    if (secElem is MessageSignature)
    {
        MessageSignature sign = secElem as MessageSignature;
        if ((sign.SignatureOptions & (SignatureOptions.IncludeSoapBody | SignatureOptions.IncludeTimestamp)) == (SignatureOptions.IncludeSoapBody | SignatureOptions.IncludeTimestamp) & sign.SigningToken is X509SecurityToken)
        {
            validSignature = true;
        }
    }
}
if (!validSignature)
    throw new SoapException("Body and Timestamp not signed in request", SoapException.ClientFaultCode);
```
Declarative security policy enforcement

WSE 2.0 also supports declarative means to either enforce (that is insert WS-Security artifacts in the SOAP header of outgoing messages) or verify (that is check for the existence of WS-Security artifacts in incoming messages) instead of writing code based on the API. WSE 2.0 SP3 uses an implementation based on the WS-SecurityPolicy specification [5] version 1.0 from December 2002, which is loosely based on the overall WS-Policy framework [6] to describe and manage Web Service related metadata.

As an example, we will use the last process step again to demonstrate the declarative verification of the Purchase Order Confirmation message based on WSE’s policy engine:

- In Visual Studio, go to the Solution Explorer and choose ‘WSE Settings 2.0’ from the context menu of the ReplenishmentService project
- Navigate to the policy configuration tab (assuming that general WSE support is already enabled for the project in the ‘General’ settings)
- To enable policy for the service, click on the ‘Enable Policy’ checkbox and add a new endpoint URI to the (yet empty) list of policies with the value of the ReplenishmentService endpoint URL (e.g. http://127.0.0.1/ReplenishmentService/ReplenishmentService.asmx)
- The WSE Security Settings wizard will be launched that will generate the policy assertions for you. As demonstrated below, the four steps to create the signature verification policy are:
  - Type of application: Secure a service application
  - Message settings: Only select the ‘Require signatures’ checkbox in the request message (and ensure that Require encryption is unselected on the response message)
  - Client Authentication Token: Choose X.509 Certificate
  - Trusted Client Certificates: Add Holiday Corp.’s public key certificate with the identifier ‘Holiday Corp. Signature’ to the list

© 2005 SAP AG

The SAP Developer Network: http://sdn.sap.com
Having a closer look at the generated policy assertions for the ReplenishmentService in the policy cache file (policyCache.config), you need to resolve the following issues regarding interoperability with SAP Web AS:

- In the `<mappings>` section, the new request policy is applied across all operations of the ReplenishmentService with the `<defaultOperation>` policy statement, even though we only want to safeguard the `confirmPurchaseOrder` operation with a digital signature. WSE also supports mapping of policies to single Web Service operations by using the `<requestAction>` policy statement. In order to use this feature, the client must provide a Web Services Addressing (WS-Addressing, [8]) header with a corresponding value in the `<wsa:Action>` element. If you want to apply different policies to operations that are accessed by a client deployed on the Web AS and thus does not implement WS-Addressing, you should consider splitting these operations into separate Web services or use the WSE 2.0 APIs as demonstrated instead of policy.

- The `<wsp:MessagePredicate>` policy assertion specifies the XML elements that must exist in the incoming SOAP message. From the default list generated by the WSE Security Settings wizard, delete all WS-Addressing related entries in the list with the namespace prefix `wsa` since the client proxy won’t send them in the request. These are `wsp:Header(wsa:To)`, `wsp:Header(wsa:Action)` and `wsp:Header(wsa:MessageID)`.

- The same issue as above also applies to the default generated list of entries in the `<wsp:MessageParts>` assertion inside the `<wssp:Integrity>` policy which defines the elements that must be signed in the incoming message. Again, only `wsp:Body()` and `wse:Timestamp()` are relevant for a Web AS client and all other entries (`wsp:Header(wsa:Action)`, `wsp:Header(wsa:FaultTo)`, `wsp:Header(wsa:From)` and all others) must be removed.
Please note that the authors of WS-SecurityPolicy published a new version 1.1 in July (2005) [7] which introduces major enhancements compared to the old version from 2002. This version will also be submitted to OASIS for standardization this fall.

**Conclusion**

WS-Security provides a flexible and future-proof mechanism to secure service-oriented enterprise applications. It is the major building block of Web Service security in the Enterprise Services Architecture (ESA) and assures interoperability across many platforms. Certainly, the flexibility comes at the price of complexity which always results in some obstacles that need to be overcome. The list below is a short summary of the key findings regarding interoperability between Web AS and the Microsoft .NET framework with an emphasis on security:

- **Use .NET WSDL command line tool to generate client proxy classes for Web AS services**: WSDL import statements (as used by all Web Services deployed on the Web AS 6.40) are not handled consistently by the build-in Visual Studio WSDL tool that generates the client proxy classes for a service via the ‘Add Web Reference’ context menu. Instead use the command line version of wsd1.exe from the .NET SDK 1.1 (e.g. C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin\wsdl.exe). Note that you have to modify the generated proxy class so that it inherits from Microsoft.Web.Services2.WebServicesClientProtocol in order to use WSE 2.0.

- **Don’t use the Digital Signature Algorithm (DSA) for signature keys**: The current implementation of the X509SecurityTokenManager in WSE 2.0 SP3 currently supports the key algorithms RSA and RSAOAEP, but not DSA for digital signatures. As the Key Storage service in Visual Administrator provides the option to choose between RSA and DSA when generating a new key, one should choose RSA for keys used to sign requests for a .NET Web Service provider.

- **Mapping of different security policies to Web Service operations**: Using the WSE API to address this issue seems to be most elegant solution unless you don’t want to setup several instances of the same service in order to use the declarative security policy for incoming request sent by Web AS clients.

- **Disable the Replay Cache in WSE**: In order to process UsernameTokens sent by Web AS client successfully, the replay cache feature in WSE must be disabled.

**References**


© 2005 SAP AG The SAP Developer Network: http://sdn.sap.com


**Download**

The following ZIP-archive contains the code for the initial (unsecured) and final VMI sample application. Please see the README.TXT file for installation instructions.

[VMII Application Code Download](#)

**Author Bio**

Martin Raepple has practiced as an Information Technology professional for over 10 years and has experience in applying technology in a wide range of industries including telecommunications, financial services, manufacturing and transport. As a Standards Architect with SAP’s Industry Standards team, Martin works in the area of standardization and interoperability testing of new Web Services technologies, focusing on message security and identity management. Martin is a frequent speaker at conferences and author of books and articles relating to information security, integration middleware and J2EE development.