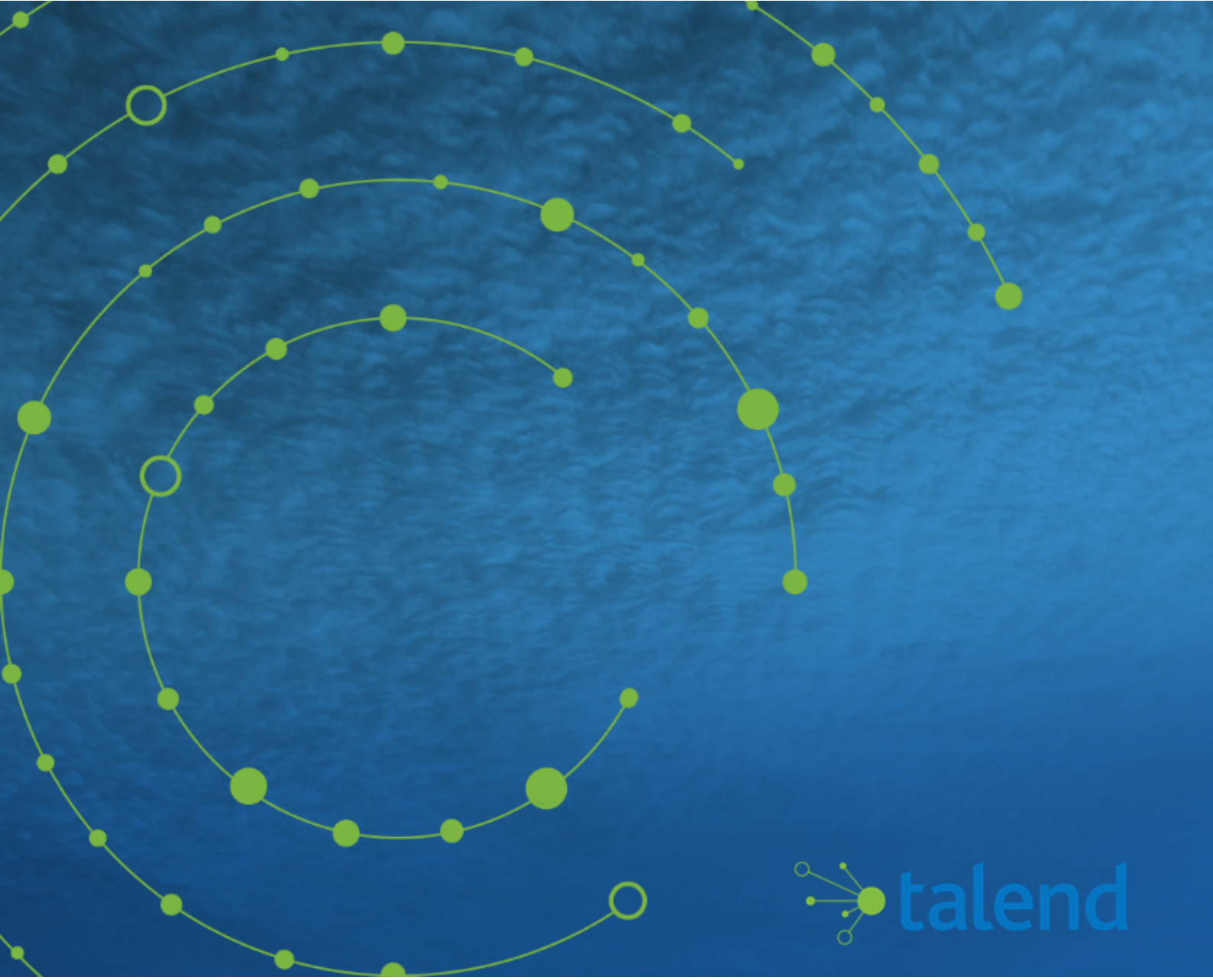


Talend ESB STS User Guide

7.2.1

Contents

Copyright

Adapted for 7.2.1. Supersedes previous releases.

Publication date: June 20, 2019

Copyright © 2019 Talend. All rights reserved.

The content of this document is correct at the time of publication.

However, more recent updates may be available in the online version that can be found on Talend Help Center.

Notices

Talend is a trademark of Talend, Inc.

All brands, product names, company names, trademarks and service marks are the properties of their respective owners.

End User License Agreement

The software described in this documentation is provided under **Talend** 's End User Software and Subscription Agreement ("Agreement") for commercial products. By using the software, you are considered to have fully understood and unconditionally accepted all the terms and conditions of the Agreement.

To read the Agreement now, visit http://www.talend.com/legal-terms/us-eula?utm_medium=help&utm_source=help_content

Security Token Service: Concepts and principles

In a heterogeneous environment, Web services need to authenticate service clients to control their access by using WS-Security (Web Services security). When negotiating trust between service clients and service providers, an authentication broker can provide a common access control infrastructure for a group of applications. Typically, the authentication broker issues signed security tokens which are used by clients to authenticate themselves at the service.

The Security Token Service is a service for providing such an authentication broker. It issues Security Tokens based on the WS-Trust, a standardized specification of Web services based on WS-Security.

This is useful, for example, to establish a trust relationship between a client and a web service, particularly if they are in different security domains. The Security Token Service is used to issue a security token, that is, a collection of claims such as name, role, and authorization code, for the client to access the service. The message receiver only must know the STS certificate for verifying the token signature to get a trusted statement of the authentication information.

What is a Security Token Service?

An informal description of a Security Token Service is that it is a web service that offers some or all of the following services (among others):

- It can issue a Security Token of some sort based on presented or configured credentials.
- It can say whether a given Security Token is valid or not.
- It can renew (extend the validity of) a given Security Token.
- It can cancel (remove the validity of) a given Security Token.
- It can transform a given Security Token into a Security Token of a different sort.

Offloading this functionality to another service greatly simplifies client and service provider functionality, as they can simply call the STS appropriately rather than have to handle the security processing logic themselves. For example, the WSDL of a service provider might state that a particular type of security token is required to access the service. Then:

1. A client of the service can ask an STS for a Security Token of that particular type, which is then sent to the service provider.
2. The service provider could choose to validate the received token locally, or dispatch the token to an STS for validation.

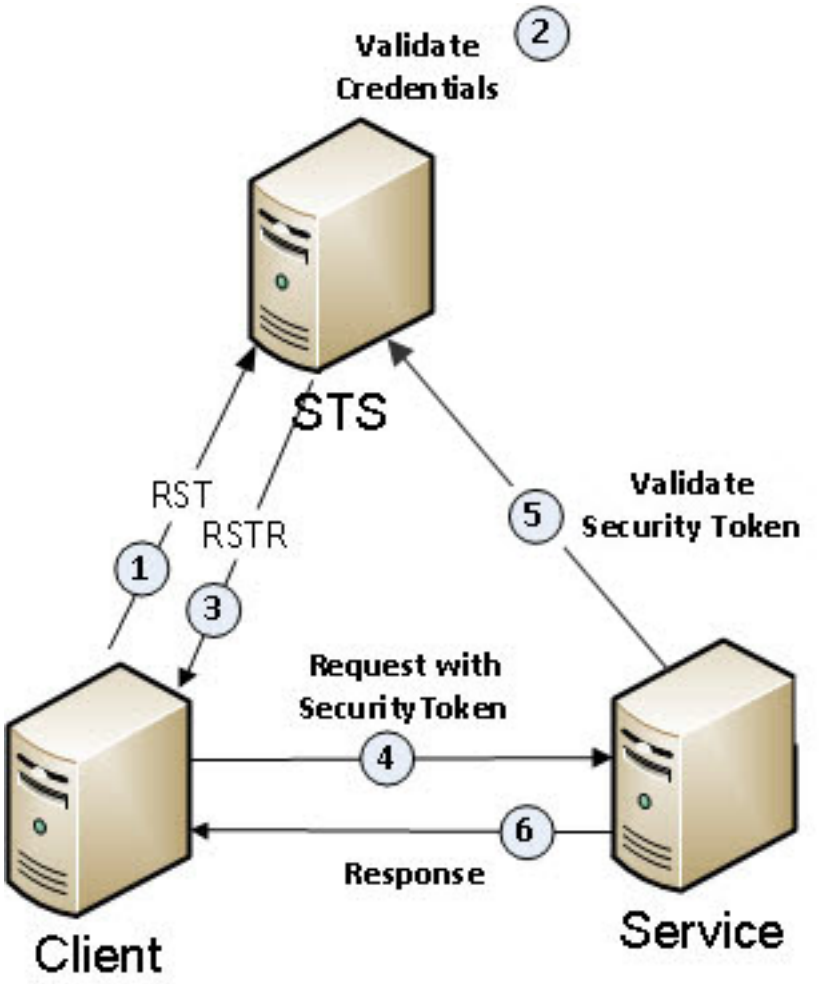
These are the two most common use cases of an STS.

For more information about the usage of Security Token Service, please read the STS chapter in the *Talend ESB Infrastructure Services Configuration Guide*.

A sample Security Token Service scenario

About this task

This section describes a typical interaction with the Security Token Service.



Procedure

1. The client sends an authentication request to the Security Token Service (Request Security Token - RST message).
2. The Security Token Service validates the client's credentials.
3. The Security Token Service issues a security token to the client (Request Security Token Response - RSTR message). The RSTR contains a security token, such as an XML Security Assertion Markup Language (SAML) token.
4. The client initializes and sends a request message, containing the token, to the Service.
5. The Service attempts to verify that the security token was issued by a trusted Security Token Service by checking the corresponding STS certificate. On success accepts it (essentially as equivalent to a "valid login"), and processes the request.
6. The service initializes and sends a response message to the client.

Results

The Security Assertion Markup Language (SAML) tokens provide cross-platform interoperability and exchange security information between clients and services in different security domains. The receiver of the message with the token only needs to know the corresponding STS certificate in order to verify the token and able to use the authentication information from the token.

STS use cases in more detail

Security Token Services are defined formally within the OASIS WS-Trust specification. They help immensely in decoupling authentication and authorization maintenance from the web service clients and providers that need that information. Using the STS eliminates the need for the Web Service Provider (WSP) and Web Service Clients (WSC) to have a direct trust relationship, freeing WSPs from needing to maintain WSC UsernameToken passwords or X509 certificates from acceptable clients. Instead, it is just necessary for the WSP to trust the STS and for the STS to be able to validate the WSC's credentials prior to making the STS call.

A client can communicate with the STS via a protocol defined in the WS-Trust specification. The SOAP Body of the request contains a "RequestSecurityToken" element as follows:

```
<wst:RequestSecurityToken Context="..." xmlns:wst="...">
   <wst:TokenType>...</wst:TokenType>
   <wst:RequestType>...</wst:RequestType>
   <wst:SecondaryParameters>...</wst:SecondaryParameters>
   ...
</wst:RequestSecurityToken>
```

The Apache CXF STS implementation supports a wide range of parameters that are passed in the RequestSecurityToken element. The SOAP Body of the response from the STS will contain a "RequestSecurityTokenResponse(Collection)" element, e.g.:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
   <wst:RequestSecurityTokenResponse>
   ...
   </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

A sample request/response for issuing a Security Token

A sample client request is given here, where the client wants the STS to issue a Security Assertion Markup Language (SAML) 2.0 token for a service hosted at `http://cxf.apache.org:8080/service`:

```
<wst:RequestSecurityToken Context="..." xmlns:wst="...">
   <wst:TokenType>
      http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0
   </wst:TokenType>
   <wst:RequestType>
      http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
   </wst:RequestType>
   <wsp:AppliesTo>http://cxf.apache.org:8080/service</wsp:AppliesTo>
</wst:RequestSecurityToken>
```

The STS responds with:

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
   <wst:RequestSecurityTokenResponse>
      <wst:TokenType>
         http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0
      </wst:TokenType>
      <wst:RequestedSecurityToken>
         <saml2:Assertion xmlns:saml2="..." ... />
      </wst:RequestedSecurityToken>
   </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

The STS provider framework in Apache CXF

The first support for an STS in Apache CXF appeared in the 2.4.0 release with the addition of an STS provider framework in the WS-Security module. This is essentially an API that can be used to create your own STS implementation. As the STS implementation shipped in CXF 2.5 is based on this provider framework, it makes sense to examine it in more detail.

The SEI (Service Endpoint Interface) is available here. It contains the following methods that are relevant to the STS features discussed above:

- `RequestSecurityTokenResponseCollectionType issue(RequestSecurityTokenType request)` - to issue a security token
- `RequestSecurityTokenResponseType issueSingle( RequestSecurityTokenType request)` - to issue a security token that is not contained in a "Collection" wrapper (for legacy applications)
- `RequestSecurityTokenResponseType cancel(RequestSecurityTokenType request)` - to cancel a security token
- `RequestSecurityTokenResponseType validate(RequestSecurityTokenType request)` - to validate a security token
- `RequestSecurityTokenResponseType renew(RequestSecurityTokenType request)` - to renew a security token

The SEI implementation handles each request by delegating it to a particular operation, which is just an interface that must be implemented by the provider framework implementation. Finally, a JAX-WS provider is available, which dispatches a request to the appropriate operation.

Security Token Service Architecture

Now we look at the Security Token Service Architecture and APIs in more detail.

- We examine the TokenProvider interface, the SCTProvider and the SAMLTokenProvider.
- We also look in detail at SAML Tokens.
- In addition, we look at the APIs involved in token handling, including validation, renewal, batch processing and cancellation.

The TokenProvider Interface

Security tokens are created in the Security Token Service via the TokenProvider interface. It has three methods:

- `boolean canHandleToken(String tokenType)` - Whether this TokenProvider implementation can provide a token of the given type
- `boolean canHandleToken(String tokenType, String realm)` - Whether this TokenProvider implementation can provide a token of the given type, in the given realm
- `TokenProviderResponse createToken(TokenProviderParameters tokenParameters)` - Create a token using the given parameters

A client can request a security token from the STS by either invoking the `issue` operation and supplying a desired token type, or else calling the "validate" operation and passing a (different) token type (token transformation). Assuming that the client request is authenticated and well-formed, the STS will iterate through a list of TokenProvider implementations to see if they can "handle" the received token type. If they can, then the implementation is used to create a security token, which is returned to the client. The second "canHandleToken" method which also takes a "realm" parameter.

So to support the issuing of a particular token type in an STS deployment, it is necessary to specify a TokenProvider implementation that can handle that token type. The STS currently ships with two TokenProvider implementations, one for generating SecurityContextTokens, and one for generating SAML Assertions. Before we look at these two implementations, let's take a look at the "createToken" operation in more detail. This method takes a TokenProviderParameters instance.

TokenProvider Parameters

The TokenProviderParameters class is nothing more than a collection of configuration properties to use in creating the token, which are populated by the STS operations using information collated from the request, or static configuration, etc. The properties of the TokenProviderParameters are:

- `STSPropertiesMBean stsProperties` - A configuration MBean that holds the configuration for the STS as a whole, such as information about the private key to use to sign issued tokens, etc.
- `EncryptionProperties encryptionProperties` - A properties object that holds encryption information relevant to the intended recipient of the token.
- `Principal principal` - The current client Principal object. This can be used as the "subject" of the generated token.
- `WebServiceContext webServiceContext` - The current web service context object. This allows access to the client request.
- `RequestClaimCollection requestedClaims` - The requested claims in the token.
- `KeyRequirements keyRequirements` - A set of configuration properties relating to keys.

- `TokenRequirements tokenRequirements` - A set of configuration properties relating to the token.
- `String appliesToAddress` - The URL that corresponds to the intended recipient of the token.
- `ClaimsManager claimsManager` - An object that can manage claims.
- `Map<String, Object> additionalProperties` - Any additional (custom) properties that might be used by a TokenProvider implementation.
- `TokenStore tokenStore` - A cache used to store tokens.
- `String realm` - The realm to create the token in (this should be the same as the realm passed to "canHandleToken").

If this looks complicated then remember that the STS will take care of populating all of these properties from the request and some additional configuration. You only need to worry about the TokenProviderParameters object if you are creating your own TokenProvider implementation.

TokenProviderResponse

The "createToken" method returns an object of type TokenProviderResponse. Similar to the TokenProviderParameters object, this just holds a collection of objects that is parsed by the STS operation to construct a response to the client. The properties are:

- `Element token` - The (DOM) token that was created by the TokenProvider.
- `String tokenId` - The ID of the token
- `long lifetime` - The lifetime of the token
- `byte[] entropy` - Any entropy associated with the token
- `long keySize` - The key size of a secret key associated with the token.
- `boolean computedKey` - Whether a computed key algorithm was used in generating a secret key.
- `TokenReference attachedReference` - An object which gives information how to refer to the token when it is "attached".
- `TokenReference unAttachedReference` - An object which gives information how to refer to the token when it is "unattached".

Most of these properties are optional as far as the STS operation is concerned, apart from the token and token ID. The TokenReference object contains information about how to refer to the token (direct reference vs. Key Identifier, etc.), that is used by the STS to generate the appropriate reference to return to the client.

The SCTProvider

Now that we've covered the TokenProvider interface, let's look at an implementation that is shipped with the STS. The SCTProvider is used to provide a token known as a SecurityContextToken, that is defined in the WS-SecureConversation specification. A SecurityContextToken essentially consists of a String Identifier which is associated with a particular secret key. If a service provider receives a SOAP message with a digital signature which refers to a SecurityContextToken in the KeyInfo of the signature, then the service provider knows that it must somehow obtain a secret key associated with that particular Identifier to verify the signature. How this is done is "out of band".

To request a SecurityContextToken, the client must use one of the following Token Types:

- http://schemas.xmlsoap.org/ws/2005/02/sc/
- http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

Two properties can be configured on the SCTProvider directly:

- `long lifetime` - The lifetime of the generated SCT. The default is 30 minutes.
- `boolean returnEntropy` - Whether to return any entropy bytes to the client or not. The default is true.

The SCTProvider generates a secret key using the KeyRequirements object that was supplied, and constructs a SecurityContextToken with a random Identifier. It creates a CXF SecurityToken object that wraps this information, and stores it in the supplied cache using the given lifetime. The SecurityContextToken element is then returned, along with the appropriate references, lifetime element, entropy, etc.

When requesting a token from an STS, the client will typically present some entropy along with a computed key algorithm. The STS will generate some entropy of its own, and combine it with the client entropy using the computed key algorithm to generate the secret key. Alternatively, the client will present no entropy, and the STS will supply all of the entropy. Any entropy the STS generates is then returned to the client, who can recreate the secret key using its own entropy, the STS entropy, and the computed key algorithm.

This secret key is then used for the SCT use-case to encrypt/sign some part of a message. The SecurityContextToken is placed in the security header of the message, and referred to in the KeyInfo element of the signed/encrypted structure. As noted earlier, the service provider must obtain somehow the secret key corresponding to the SecurityContextToken identifier. Perhaps the service provider shares a (secured) distributed cache with an STS instance. Or perhaps the service provider sends the SCT to an STS instance to "validate" it, and receives a SAML token in response with the embedded (encrypted) secret key.

TokenProvider token caching

Finally, we will cover token caching in a TokenProvider implementation. The SCTProvider is essentially useless without a cache, as otherwise there is no way for a third-party to know the secret key corresponding to a SecurityContextToken. Any TokenProvider implementation can cache a generated token in the TokenStore object supplied as part of the TokenProviderParameters.

The SCTProvider creates a SecurityToken with the ID of the SCT, the secret key associated with the SCT and the client principal. If a "realm" is passed through, then this is recorded as a property of the SecurityToken (keyed via STSConstants.TOKEN_REALM). Finally, the STS ships with two TokenStore implementations, an in-memory implementation based on eh-cache, and an implementation that uses Hazelcast.

The SAMLTokenProvider

The SAMLTokenProvider can issue SAML 1.1 and SAML 2.0 tokens. To request a SAML 1.1 token, the client must use one of the following Token Types:

- http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1
- urn:oasis:names:tc:SAML:1.0:assertion

To request a SAML 2.0 token, the client must use one of the following Token Types:

- http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0

- urn:oasis:names:tc:SAML:2.0:assertion

The following properties can be configured on the SAMLTokenProvider directly:

- List<AttributeStatementProvider> attributeStatementProviders - A list of objects that can add attribute statements to the token.
- List<AuthenticationStatementProvider> authenticationStatementProviders - A list of objects that can add authentication statements to the token.
- List<AuthDecisionStatementProvider> authDecisionStatementProviders - A list of objects that can add authorization decision statements to the token.
- SubjectProvider subjectProvider - An object used to add a Subject to the token.
- ConditionsProvider conditionsProvider - An object used to add a Conditions statement to the token.
- boolean signToken - Whether to sign the token or not. The default is true.
- Map<String, SAMLRealm> realmMap - A map of realms to SAMLRealm objects.

We will explain each of these properties in more detail in the next few sections.

Realms in the Token Providers

As explained in the previous section, the TokenProvider interface has a method that takes a realm parameter:

- `boolean canHandleToken(String tokenType, String realm)` - Whether this TokenProvider implementation can provide a token of the given type, in the given realm

In other words, the TokenProvider implementation is being asked whether it can supply a token corresponding to the Token Type in a particular realm. How the STS knows what the desired realm is will be covered subsequently. However, we will explain how the realm is handled by the TokenProviders here. The SCTProvider ignores the realm in the canHandleToken method. In other words, the SCTProvider can issue a SecurityContextToken in any realm. If a realm is passed through via the TokenProviderParameters when creating the token, the SCTProvider will cache the token with the associated realm as a property.

Unlike the SCTProvider, the SAMLTokenProvider does not ignore the realm parameter to the `canHandleToken` method. Recall that the SAMLTokenProvider has a property "Map<String, SAMLRealm> realmMap". The `canHandleToken` method checks to see if the given realm is null, and if it is not null then the realmMap must contain a key which matches the given realm. So if the STS implementation is designed to issue tokens in different realms, then the realmMap of the SAMLTokenProvider must contain the corresponding realms in the key-set of the map.

The realmMap property maps realm Strings to SAMLRealm objects. Among other properties, the SAMLRealm class contains the following settings:

- String issuer - the Issuer String to use in this realm
- String signatureAlias - the keystore alias to use to retrieve the private key the SAMLTokenProvider uses to sign the generated token

If the SAMLTokenProvider is "realm aware", then it can issue tokens with an issuer name and signing key specific to a given realm. SAMLRealms also contain cryptographic and CallbackHandle r settings to allow for configuring realm-specific keystores if desired. If no realm is passed to the SAMLTokenProvider, then these properties are obtained from the "system wide" properties defined in the STSPropertiesMBean object passed as part of the TokenProviderParameters, which can be set via the following methods:

- `void setSignatureUsername(String signatureUsername)`
- `void setIssuer(String issuer)`

Two additional properties are required when signing SAML Tokens. A password is required to access the private key in the keystore, which is supplied by a CallbackHandler instance. A WSS4J "Crypto" instance is also required which controls access to the keystore. These are both set on the STSPropertiesMBean object via:

- `void setCallbackHandler(CallbackHandler callbackHandler)`
- `void setSignatureCrypto(Crypto signatureCrypto)`

Note that the signature of generated SAML Tokens can be disabled, by setting the "signToken" property of the SAMLTokenProvider to "false". As per the SCTProvider, the generated SAML tokens are stored in the cache with the associated realm stored as a property.

Populating SAML Tokens

In the previous section we covered how a generated SAML token is signed, how to configure the key used to sign the assertion, and how to set the Issuer of the Assertion. In this section we will describe how to populate the SAML Token itself. The SAMLTokenProvider is designed to be able to issue a wide range of SAML Tokens. It does this by re-using the SAML abstraction library that ships with Apache WSS4J, which defines a collection of beans that are configured and then assembled in a CallbackHandler to create a SAML assertion.

Configure a Conditions statement

The SAMLTokenProvider has a "ConditionsProvider conditionsProvider" property, which can be used to configure the generated Conditions statement which is added to the SAML Assertion. The ConditionsProvider has a method to return a ConditionsBean object, and a method to return a lifetime in seconds. The ConditionsBean holds properties such as the not-before and not-after dates, etc. The SAMLTokenProvider ships with a default ConditionsProvider implementation that is used to insert a Conditions statement in every SAML token that is generated. This implementation uses a default lifetime of 30 minutes, and set the Audience Restriction URI of the Conditions Statement to be the received "AppliesTo" address, which is obtained from the TokenProviderParameters object.

The DefaultConditionsProvider can be configured to change the lifetime of the issued token. If you want to remove the ConditionsProvider altogether from the generation assertion, or implement a custom Conditions statement, then you must implement an instance of the ConditionsProvider interface, and set it on the SAMLTokenProvider.

Configure a Subject

The SAMLTokenProvider has a "SubjectProvider subjectProvider" property, which can be used to configure the Subject of the generated token, regardless of the version of the token. The SubjectProvider interface defines a single method to return a SubjectBean, given the token provider parameters, the parent Document of the assertion, and a secret key to use (if any). The SubjectBean contains the Subject name, name-qualifier, confirmation method, and KeyInfo element, amongst other properties. The SAMLTokenProvider ships with a default SubjectProvider implementation that is used to insert a Subject into every SAML Token that is generated.

The DefaultSubjectProvider has a single configuration method to set the subject name qualifier. It creates a subject confirmation method by checking the received key type. The subject name is the

name of the principal obtained from TokenProviderParameters. Finally, a KeyInfo element is set on the SubjectBean under the following conditions:

- If a "SymmetricKey" Key Type algorithm is specified by the client, then the secret key passed through to the SubjectProvider is encrypted with the X509Certificate of the recipient, and added to the KeyInfo element. How the provider knows the public key of the recipient will be covered subsequently.
- If a "PublicKey" KeyType algorithm is specified by the client, the X509Certificate that is received as part of the "UseKey" request is inserted into the KeyInfo element of the Subject.

If a "Bearer" KeyType algorithm is specified by the client, then no KeyInfo element is added to the Subject. For the "SymmetricKey" Key Type case, the SAMLTokenProvider creates a secret key using a SymmetricKeyHandler instance. The SymmetricKeyHandler first checks the key size that is supplied as part of the KeyRequirements object, by checking that it fits in between a minimum and maximum key size that can be configured. It also checks any client entropy that is supplied, as well as the computed key algorithm. It then creates some entropy and a secret key.

To add a custom Subject element to an assertion, you must create your own SubjectProvider implementation, and set it on the SAMLTokenProvider.

Adding Attribute Statements

The SAMLTokenProvider has a "List<AttributeStatementProvider> attributeStatementProviders" property, which can be used to add AttributeStatments to the generated assertion. Each object in the list adds a single Attribute statement. The AttributeStatementProvider contains a single method to return an AttributeStatementBean given the TokenProviderParameters object. This contains a SubjectBean (for SAML 1.1 assertions), and a list of AttributeBeans. The AttributeBean object holds the attribute name/qualified-name/name-format, and a list of attribute values, amongst other properties.

If no statement provider is configured in the SAMLTokenProvider, then the DefaultAttributeStatementProvider is invoked to create an Attribute statement to add to the assertion. It creates a default "authenticated" attribute, and also creates separate Attributes for any "OnBehalfOf" or "ActAs" elements that were received in the request. If the received OnBehalfOf/ActAs element was a UsernameToken, then the username is added as an Attribute. If the received element was a SAML Assertion, then the subject name is added as an Attribute.

Adding Authentication Statements

The SAMLTokenProvider has a "List<AuthenticationStatementProvider> authenticationStatementProviders" property, which can be used to add AuthenticationStatements to the generated assertion. Each object in the list adds a single Authentication statement. The AuthenticationStatementProvider contains a single method to return an AuthenticationStatementBean given the TokenProviderParameters object. This contains a SubjectBean (for SAML 1.1 assertions), an authentication instant, authentication method, and other properties. No default implementation of the AuthenticationStatementProvider interface is provided in the STS, so if you want to issue Authentication Statements you will have to write your own.

Adding Authorization Decision Statements

The SAMLTokenProvider has a "List<AuthDecisionStatementProvider> authDecisionStatementProviders" property, which can be used to add AuthzDecisionStatements to the generated assertion. Each object in the list adds a single statement. The AuthDecisionStatementProvider contains a single method to return an AuthDecisionStatementBean given the TokenProviderParameters object. This

contains a SubjectBean (for SAML 1.1 assertions), the decision (permit/indeterminate/deny), the resource URI, a list of ActionBeans, amongst other properties. No default implementation of the AuthDecisionStatementProvider interface is provided in the STS.

Note that for SAML 1.1 tokens, the Subject is embedded in one of the Statements. When creating a SAML 1.1 Assertion, if a given Authentication/Attribute/AuthzDecision statement does not have a subject, then the standalone Subject is inserted into the statement. Finally, once a SAML token has been created, it is stored in the cache (if one is configured), with a lifetime corresponding to that of the Conditions statement. A TokenProviderResponse object is created with the DOM representation of the SAML Token, the SAML Token ID, lifetime, entropy bytes, references, etc.

Token Validation

The TokenValidator interface

SecurityTokens are validated in the STS via the TokenValidator interface. It is very similar to the TokenProvider interface. It has three methods:

- `boolean canHandleToken(ReceivedToken validateTarget)` - Whether this TokenValidator implementation can validate the given token
- `boolean canHandleToken(ReceivedToken validateTarget, String realm)` - Whether this TokenValidator implementation can validate the given token in the given realm
- `TokenValidatorResponse validateToken(TokenValidatorParameters tvp)` - Validate a token using the given parameters.

A client can validate a security token via the STS by invoking the "validate" operation. Assuming that the client request is authenticated and well-formed, the STS will iterate through a list of TokenValidator implementations to see if one can "handle" the received token. If one can, then that implementation is used to validate the received security token, and the validation result is returned to the client. The second "canHandleToken" method also takes a "realm" parameter.

So to support the validation of a particular token type in an STS deployment, it is necessary to specify a TokenValidator implementation that can handle that token. The STS currently ships with four TokenValidator implementations, to validate SecurityContextTokens, SAML Assertions, UsernameTokens, and BinarySecurityTokens. Before we look at these implementations, let's take a look at the "validateToken" operation in more detail. This method takes a TokenValidatorParameters instance.

TokenValidatorParameters

The TokenValidatorParameters class is a collection of configuration properties to use in validating the token, which are populated by the STS operations using information collated from the request, or static configuration, etc. The properties of the TokenValidatorParameters are:

- STSPropertiesMBean stsProperties - A configuration MBean that holds the configuration for the STS as a whole.
- Principal principal - The current client Principal object
- WebServiceContext webServiceContext - The current web service context object. This allows access to the client request.
- KeyRequirements keyRequirements - A set of configuration properties relating to keys.
- TokenRequirements tokenRequirements - A set of configuration properties relating to the token.
- TokenStore tokenStore - A cache used to retrieve tokens.

- String realm - The realm to validate the token in (this should be the same as the realm passed to "canHandleToken").
- ReceivedToken token - Represents the token that was received for validation.

If this looks complicated then remember that the STS will take care of populating all of these properties from the request and some additional configuration. You only need to worry about the TokenValidatorParameters object if you are creating your own TokenValidator implementation.

TokenValidatorResponse

The "validateToken" method returns an object of type TokenValidatorResponse. Similar to the TokenValidatorParameters object, this just holds a collection of objects that is parsed by the STS operation to construct a response to the client. The properties are:

- ReceivedToken token - Represents the token that was received for validation. If the token is determined valid, then the ReceivedToken will be given a valid state ReceivedToken.STATE.VALID, otherwise STATE.INVALID or STATE.EXPIRED
- Principal principal - A principal corresponding to the validated token.
- Map<String, Object> additionalProperties - Any additional properties associated with the validated token.
- String realm - The realm of the validated token.

The SCTValidator

Now that we've covered the TokenValidator interface, let's look at an implementation that is shipped with the STS. The SCTValidator is used to validate a token known as a SecurityContextToken, that is defined in the WS-SecureConversation specification. The SCTProvider was covered in earlier in this chapter. A SecurityContextToken essentially consists of a String Identifier which is associated with a particular secret key. If a service provider receives a SOAP message with a digital signature which refers to a SecurityContextToken in the KeyInfo of the signature, then the service provider knows that it must somehow obtain a secret key associated with that particular Identifier to verify the signature.

One way to do this would be if the service provider shares a (secured) distributed cache with an STS instance. An alternative solution would be for the service provider to send the SCT to an STS for validation, and to receive a SAML token in response with the embedded (encrypted) secret key. The SCTValidator can accommodate this latter scenario, albeit indirectly as will be explained shortly.

The SCTValidator can validate a SecurityContextToken in either of the following namespaces:

- http://schemas.xmlsoap.org/ws/2005/02/sc/
- http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

The SCTValidator validates a received SecurityContextToken by checking to see whether it is stored in the cache. Therefore it is a requirement to configure a cache for the STS if you want to validate SecurityContextTokens. If the SecurityContextToken is stored in the cache (for example, by the SCTProvider), then the received SecurityToken is taken to be valid. The secret associated with the SecurityContextToken is also retrieved from the cache, and set as an "additional property" in the TokenValidatorResponse using the key "sct-validator-secret". If the cached token has a stored principal, then this is also returned in the TokenValidatorResponse.

If you want to support the scenario of returning the secret key associated with the SecurityContex tToken to the client (of the STS), then it is possible to do so via token transformation. This is where the client sends an additional Token Type (in this case for a SAML Token). After the token is validated, the SAMLTokenProvider is called with the additional properties map obtained from the SCTValidator.

The SAMLTokenProvider then has access to the secret key via the "sct-validator-secret" tag, which it can insert into the Assertion using a custom AttributeProvider.

The X509TokenValidator

Another TokenValidator implementation that ships with the STS is the X509TokenValidator. This class validates an X.509 V.3 certificate (received as a BinarySecurityToken). The BinarySecurityToken must use Base-64 encoding. The received cert must be known (or trusted) by the STS crypto object, that is set on the STSPropertiesMBean object. The X509TokenValidator has a single property that can be configured:

- `void setValidator(Validator validator)` - Set the WSS4J Validator instance to use to validate the received certificate. The default is SignatureTrustValidator.

No proof-of-possession is done with the received certificate. The subject principal of the certificate is set on the response, if validation is successful. Note that no caching is used in this TokenValidator implementation.

The UsernameTokenValidator

The UsernameTokenValidator is used to validate WS-Security UsernameTokens. Two properties can be set directly on the UsernameTokenValidator:

- `void setValidator(Validator validator)` - Set the WSS4J Validator instance to use to validate the received UsernameToken. The default is the UsernameTokenValidator (note that this is in WSS4J and not the same as the UsernameTokenValidator in the STS!).
- `void setUsernameTokenRealmCodec(UsernameTokenRealmCodec uTRC)` - Set the UsernameTokenRealmCodec instance to use to return a realm from a validated token.

The UsernameToken is first checked to make sure that it is well-formed. If it has no password element then it is rejected. If a cache is configured, then it sees if the UsernameToken has been previously stored in the cache (searching by wsu:Id). If it is, the CXF STS checks the wsu:Id-indepen dent hashcode of the UsernameToken, and searches for the String representation of this hash in the TokenStore. If a SecurityToken is retrieved, a second check that the stored hash of the SecurityToken matches the hash of the received token. Note that the CXF STS does not have a UsernameTokenP rovider as of yet, so for this use-case perhaps the cache is shared with a custom TokenProvider.

If the token is not stored in the cache, then the WSS4J Validator instance is used to validate the received UsernameToken. As stated above, the default implementation that is used is the UsernameTokenValidator in WSS4J. This implementation uses a CallbackHandler to supply a password to validate the UsernameToken. This CallbackHandler implementation is supplied by the STSPropertiesMBean object. WSS4J also ships with an implementation that validates a UsernameToken via a JAAS LoginModule, which can be plugged in to the STS UsernameTokenV alidator. If validation is successful, then a principal is created from the received UsernameToken and set on the response.

Realms in the TokenValidators

Recall that the TokenValidator interface has a method that takes a realm parameter:

- `boolean canHandleToken(ReceivedToken validateTarget, String realm)` - Whether this TokenValidator implementation can validate the given token in the given realm

Realms are handled in a slightly different way in TokenValidators compared to TokenProviders. Recall that for TokenProviders, the implementation is essentially asked whether it can provide a token in a given realm. For the SCTProvider, the realm is ignored in this method. However, when creating a token, the SCTProvider will store the given realm as a property associated with that token in the cache. The SAMLTokenProvider checks to see if the given realm is null, and if it is not null then the realmMap must contain a key which matches the given realm.

There is a subtle distinction between the realm passed to "canHandleToken" for TokenValidators and the realm returned after a token is validated as part of the TokenValidatorResponse object. The realm passed to "canHandleToken" is the realm to validate the token in. So for example, you could have two TokenValidator instances registered to validate the same token, but in different realms. All of the TokenValidator implementations that ship with the STS ignore the realm as part of this method. However, the method signature gives the user the option to validate tokens in different realms in a more flexible manner.

The realm that is returned as part of the TokenValidatorResponse is the realm that the validated token is in (if any). This can be different to the realm the token was validated in. The X509TokenValidator ignores this parameter altogether. The SCTValidator checks to see whether the SecurityToken that was stored in the cache has a realm property, and if so sets this on the TokenValidatorResonse. The UsernameTokenValidator and SAMLTokenValidator handle realms in a more sophisticated manner. Recall that the UsernameTokenValidator has the following method:

- `void setUsernameTokenRealmCodec(UsernameTokenRealmCodec uTRC)` - Set the UsernameTokenRealmCodec instance to use to return a realm from a validated token.

The UsernameTokenRealmCodec has a single method:

- `String getRealmFromToken(UsernameToken usernameToken)` - Get the realm associated with the UsernameToken parameter.

No UsernameTokenRealmCodec implementation is set by default on the UsernameTokenValidator, hence no realm is returned in TokenValidatorResponse. If an implemention is specified, then the UsernameTokenValidator will retrieve a realm from the UsernameTokenRealmCodec implementation corresponding to the validated UsernameToken. If a cache is configured, and the UsernameToken was already stored in the cache, then the realm is compared to the realm of the cached token, stored under the tag `org.apache.cxf.sts.token.realm`. If they do not match then validation fails.

The SAMLTokenValidator

The SAMLTokenValidator is used to validate SAML (1.1 and 2.0) tokens. The following properties can be set directly on the SAMLTokenValidator:

- `void setValidator(Validator validator)` - Set the WSS4J Validator instance to use to validate the received certificate. The default is SignatureTrustValidator.
- `void setSamlRealmCodec(SAMLRealmCodec samlRealmCodec)` - Set the SAMLRealmCodec instance to use to return a realm from a validated token.
- `void setSubjectConstraints(List<String> subjectConstraints)` - Set a list of Strings corresponding to regular expression constraints on the subject DN of a certificate that was used to sign an Assertion.

These methods are covered in more detail below. The Assertion is first checked to make sure that it is well-formed. If a cache is defined, then the hashcode of the Assertion is checked against the hash of all assertions in the cache. If a match is found in the cache, then the Assertion is taken to be valid. If a match is not found, then the Assertion is validated.

Validating a received SAML Assertion

If the token is not stored in the cache then it must be validated. Firstly a check is performed to make sure that the Assertion is signed, if it is not then it is rejected. The signature of the Assertion is then validated using the Crypto object retrieved from the STSPropertiesMBean passed in the TokenValidatorParameters. Finally, trust is verified in the certificate/public-key used to sign the Assertion. This is done using the Validator object that can be configured via "setValidator". The default Validator is the WSS4J SignatureTrustValidator, which checks that the received certificate is known (or trusted) by the STS Crypto object.

Recall that a List of Strings can be set on the SAMLTokenValidator via the "setSubjectConstraints" method. These Strings correspond to regular expression constraints on the subject DN of a certificate that was used to sign an Assertion. This provides additional flexibility to validate a received SAML Assertion. For example, the Assertion could be signed by an entity that has a certificate issued by a particular CA, which in turn is trusted by the STS Crypto object. However, one might want to restrict the list of "valid" entities who can sign a SAML Assertion. This can be done by adding a list of regular expressions that match the Subject DN of all acceptable certificates that might be used to sign a valid SAML Assertion. This matching is done by the CertConstraintsParser.

Realm handling in the SAMLTokenValidator

Recall that the SAMLTokenValidator has the following method:

- `void setSamlRealmCodec(SAMLRealmCodec samlRealmCodec)` - Set the SAMLRealmCodec instance to use to return a realm from a validated token.

The SAMLRealmCodec has a single method:

- `String getRealmFromToken(AssertionWrapper assertion)` - Get the realm associated with the (SAML Assertion) parameter.

No SAMLRealmCodec implementation is set by default on the SAMLTokenValidator, hence no realm is returned in TokenValidatorResponse. If an implemention is specified, then the SAMLTokenValid ator will retrieve a realm from theSAMLRealmCodec implementation corresponding to the validated Assertion. If a cache is configured, and the Assertion was already stored in the cache, then the realm is compared to the realm of the cached token, stored under the tag "org.apache.cxf.sts.token.realm". If they do not match then validation fails.

Token Renewal

The TokenRenewer interface

Security tokens are renewed in the STS via the TokenRenewer interface. It has the following methods:

- `void setVerifyProofOfPossession(boolean verifyProofOfPossession)` - A boolean switch to enable or disable the proof of possession requirement.
- `void setAllowRenewalAfterExpiry(boolean allowRenewalAfterExpiry)` - A switch to enable or disable the ability to renew tokens after they have expired.
- `boolean canHandleToken(ReceivedToken renewTarget)` - Whether this TokenRenewer implementation can renew the given token.
- `boolean canHandleToken(ReceivedToken renewTarget, String realm)` - Whether this TokenRenewer implementation can renew the given token in the given realm.
- TokenRenewerResponse renewToken(TokenRenewerParameters tokenParameters) - Renew the token using the given parameters

A client can request that the STS renew a security token by invoking the "renew" operation and supplying a token under the "RenewTarget" Element. Assuming that the client request is authenticated and well-formed, the STS will first iterate through a list of TokenValidator implementations to see if they can "handle" the received token. If they can, then the implementation is used to validate the received security token. If no TokenValidator is found that can handle the RenewTarget that was received, then an exception is thrown. Note that this means that for token renewal, it is necessary to configure both a TokenValidator and TokenRenewer implementation that can handle the given token.

After the successful validation of a token, the state of the token is checked. If the state is not valid or expired, then an exception is thrown. The STS then iterates through the configured list of TokenRenewer implementations to see which can renew the given (validated) token. The token is then renewed and returned to the client.

The TokenRenewerParameters class is nothing more than a collection of configuration properties to use in renewing the token, which are populated by the STS operations using information collated from the request, or static configuration, etc. The TokenRenewerResponse class holds the results from the (successful) token renewal, including the DOM representation of the renewed token, the token Id, the new lifetime of the renewed token, and references to the renewed token.

The SAMLTokenRenewer

The SAMLTokenRenewer can renew valid or expired SAML 1.1 and SAML 2.0 tokens. The following properties can be configured on the SAMLTokenRenewer directly:

- `boolean signToken` - Whether to sign the renewed token or not. The default is true.
- `ConditionsProvider conditionsProvider` - An object used to add a Conditions statement to the token.
- `Map<String, SAMLRealm>` realmMap - A map of realms to SAMLRealm objects.
- `long maxExpiry` - how long a token is allowed to be expired (in seconds) before renewal. The default is 30 minutes.

The SAMLTokenRenewer first checks that the token it extracts from the TokenRenewerParameters is in an expired or valid state, if not it throws an exception. It then retrieves the cached token that corresponds to the token to be renewed. A cache must be configured to use the SAMLTokenRenewer, and the token to be renewed must be in the cache before renewal takes place, for reasons that will become clear in the next section.

Validating the token

Before the received SAML token can be renewed, a number of validation steps (that are specific to renewing SAML tokens) takes place. Two boolean properties are retrieved from the properties of the cached token:

- `org.apache.cxf.sts.token.renewing.allow` - Whether the token is allowed to be renewed or not.
- `org.apache.cxf.sts.token.renewing.allow.after.expiry` - Whether the token is allowed to be renewed or not after it has expired.

These two properties are set in the SAMLTokenProvider based on a received <wst:Renewing/> element when the user is requesting a SAML token via the issue binding. If a user omits a <wst:Renewing/> element, or sends <wst:Renewing/> or <wst:Renewing Allow="true"/>, then the token is allowed to be renewed. However, only if the user sends <wst:Renewing OK="true"/>, will the token be allowed to be renewed after expiry. This explains why a TokenStore is required for token

renewal, as without access to these two properties it is impossible for the SAMLTokenRenewer to figure out whether the issuer of the token intended for the token to be renewed (after expiry) or not.

If the state of the token is expired, and if the token is allowed to be renewed after expiry, a final check is done against the boolean set via the `setAllowRenewalAfterExpiry` method of TokenRenewer. If this is set to false (the default), then an exception is thrown. So to support token renewal after expiry, you must explicitly define this behavior on the TokenRenewer implementation . Finally, a check is done on how long ago the SAML Token expired. If it is greater than the value configured in the `maxExpiry` property (30 minutes by default), then an exception is thrown.

The next validation step is to check proof of possession, if this is enabled (true by default). The Subject KeyInfo of the Assertion must contain a PublicKey or X509Certificate that corresponds to either the client certificate if TLS is used, or to the private key that was used to sign some part of the request. Finally, if an `AppliesTo` URI is sent as part of the request, the SAMLTokenRenewer checks that the received Assertion contains at least one AudienceRestrictionURI that matches that address, otherwise it throws an Exception.

Renewing the SAML Assertion

After the validation steps outlined above have passed, the token is renewed in the following way:

- A new ID is generated for the token.
- A new `IssueInstant` is set on the token.
- A new Conditions Element replaces the old Conditions Element of the token, using the configured ConditionsProvider.
- The Assertion is (re)-signed if the `signToken` property is true.

The old token is removed from the cache, and the new token is added. Finally, the token is set on the TokenRenewerResponse, along with the token Id, and Lifetime.

SAML Token Renewal in action

Finally, let's take a look at a system test in CXF that shows how to renew a SAML Token issued by an STS. The wsdl of the service provider defines a number of endpoints which use the transport binding, with a (endorsing) supporting token requirement which has an IssuedToken policy that requires a SAML token. In other words, the client must request a SAML token from an STS and send it to the service provider over TLS, and optionally use the secret associated with the SAML token to sign the message Timestamp (if an EndorsingSupportingToken policy is specified in the wsdl).

The STS spring configuration is available here. The SAMLTokenRenewer is configured with proof-of-possession enabled, and tokens are allowed to be renewed after they have expired. Let's look at the test code and client configuration. All of the tests follow the same pattern. The client requests a SAML Token from the STS (as per the IssuedToken policy), with a TTL (time-to-live) value of 8 seconds. The client then uses this issued token to make a successful request to the service provider. The test code then sleeps for 8 seconds to expire the token, and tries to invoke on the service provider again. The IssuedTokenInterceptorProvider in the WS-Security runtime in CXF recognizes that the token has expired, and sends it to the STS for renewal. The returned (renewed) token is then sent to the service provider.

Token Batch Processing

The STS implementation in CXF is based on the STS Provider framework in the security runtime, which is essentially an API that can be used to create your own STS implementation. The SEI (Service Endpoint Implementation) contains the following method that can be used for batch processing:

- RequestSecurityTokenResponseCollectionType requestCollection(RequestSecurityTokenCollectionType requestCollection)

This method can be used to execute batch processing for any of the core operations (issue/validate/renew/cancel). To do this it is necessary to implement the RequestCollectionOperation interface, and to install it in the STS Provider.

Batch Processing in the STS implementation

The STS ships with an implementation of the RequestCollectionOperation interface described above that can be used to perform batch processing. The TokenRequestCollectionOperation is essentially a wrapper for the other operations, and does no processing itself. It iterates through the request collection that was received, and checks that each request has the same RequestType. If not then an exception is thrown. It then dispatches each request to the appropriate operation. To support bulk processing for each individual operation, it is necessary to set the appropriate implementation for that operation on the TokenRequestCollectionOperation, otherwise an exception will be thrown.

Batch Processing example

Take a look at the following test to see how batch processing works in practice. In this test, the client requests two tokens via the (batch) issue binding, a SAML 1.1 and a SAML 2.0 token. The client then validates both tokens at the same time using the batch validate binding. The STSClient class used by the WS-Security runtime in CXF does not currently support bulk processing. Therefore, the test uses a custom STSClient implementation for this purpose.

The WSDL the STS uses two separate bindings for issue and validate, to cater for the fact that two separate SOAP Actions must be used for bulk issue and validate for the same operation. The STS configuration is available here. Note that the TokenRequestCollectionOperation is composed with the TokenIssueOperation and TokenValidateOperation, to be able to bulk issue and validate security tokens:

```
<bean
   class="org.apache.cxf.sts.operation.TokenRequestCollectionOperation"
   id="transportRequestCollectionDelegate">
   <property name="issueSingleOperation" ref="transportIssueDelegate">
   <property name="validateOperation" ref="transportValidateDelegate>
</bean>
```

Token Cancellation

The TokenCanceller interface

SecurityTokens are cancelled in the STS via the TokenCanceller interface. This interface is very similar to the TokenProvider and TokenValidator interfaces. It contains three methods:

- `void setVerifyProofOfPossession(boolean verifyProofOfPossession)` - Whether to enable or disable proof-of-possession verification.
- `boolean canHandleToken(ReceivedToken cancelTarget)` - Whether this TokenCanceller implementation can cancel the given token
- `TokenCancellerResponse cancelToken(TokenCancellerParameters tokenParameters)` - Cancel a token using the given parameters

A client can cancel a security token via the STS by invoking the "cancel" operation. Assuming that the client request is authenticated and well-formed, the STS will iterate through a list of TokenCanceller

implementations to see if they can "handle" the received token. If they can, then the implementation is used to cancel the received security token, and the cancellation result is returned to the client. The STS currently ships with a single TokenCanceller implementation, which can cancel SecurityContex tTokens that were issued by the STS. Before we look at this implementation, let's look at the "cancelToken" operation in more detail. This method takes a TokenCancellerParameters instance, and returns a TokenCancellerResponse object.

TokenCancellerParameters and TokenCancellerResponse

The TokenCancellerParameters class is nothing more than a collection of configuration properties to use in cancelling the token, which are populated by the STS operations using information collated from the request, or static configuration, etc. The properties of the TokenCancellerParameters are:

- STSPropertiesMBean stsProperties - A configuration MBean that holds the configuration for the STS as a whole.
- Principal principal - The current client Principal object
- WebServiceContext webServiceContext - The current web service context object. This allows access to the client request.
- KeyRequirements keyRequirements - A set of configuration properties relating to keys.
- TokenRequirements tokenRequirements - A set of configuration properties relating to the token.
- TokenStore tokenStore - A cache used to retrieve tokens.
- ReceivedToken token - Represents the token that was received for cancellation.

The "cancelToken" method returns an object of type TokenCancellerResponse. Similar to the TokenCancellerParameters object, this just holds a collection of objects that is parsed by the STS operation to construct a response to the client. It currently only has a single property:

- ReceivedToken token - Represents the token that was received for cancellation. Its state will be STATE.CANCELLED if token cancellation was successful.

The SCTCanceller

The STS ships with a single implementation of the TokenCanceller interface, namely the SCTCanceller. The SCTCanceller is used to cancel a token known as a SecurityContextToken, that is defined in the WS-SecureConversation specification. The SCTProvider and the SCTValidator were covered previously. A SecurityContextToken essentially consists of a String Identifier which is associated with a particular secret key. The SCTCanceller can cancel a SecurityContextToken in either of the following namespaces:

- http://schemas.xmlsoap.org/ws/2005/02/sc/
- http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

Recall that the SCTValidator validates a received SecurityContextToken by checking to see whether it is stored in the cache. Therefore it is a requirement to configure a cache for the STS if you want to validate SecurityContextTokens. The same applies for the SCTCanceller. A received SecurityContextToken is successfully cancelled only if it is stored in the cache and is removed from the cache without any errors. This generally implies that the STS must have previously issued the SecurityContextToken and stored it in the cache, unless the STS is sharing a distributed cache with other STS instances.

Enforcing proof-of-possession

Recall that the TokenCanceller interface has a method "setVerifyProofOfPossession" which defines whether proof-of-possession is required or not to cancel a security token. The default value for the SCTCanceller is "true".

This means that for the client to successfully cancel a SecurityContextToken it must prove to the STS that it knows the secret key associated with that SecurityContextToken. The client must do this by signing some portion of the request with the same secret key that the SCTCanceller retrieves from the security token stored in the cache.

Token Caching

This section covers how security tokens are cached in the CXF WS-Security runtime and also in the STS.

CXF WS-Security runtime token caching

CXF caches tokens in the security runtime in the following circumstances:

- When the IssuedTokenInterceptorProvider is invoked to obtain an Issued token from an STS.
- When the STSTokenValidator is used to validate a received UsernameToken, BinarySecurityToken or SAML Assertion to an STS.
- When the SecureConversation protocol is used.
- When the WS-Trust SPNEGO protocol is used.
- When tokens are obtained from a Kerberos KDC.

In each of these use-cases, the retrieved token is cached to prevent repeated remote calls to obtain the desired security token. There is no built-in support as yet to cache tokens in the WS-Security layer to prevent repeat validation. Of course this could be easily done by wrapping the existing validators with a custom caching solution.

SecurityTokens

CXF defines a SecurityToken class which encapsulates all relevant information about a successful authentication event in the security runtime (as defined above). In particular, it contains the following items (among others):

- A String identifier of the token. This could be a SAML Assertion Id, the Identifier element of a SecurityContextToken, or the wsu:Id of a UsernameToken, etc.
- The DOM Element that represents that security token.
- Attached and Unattached reference elements for that token that might have been retrieved from an STS.
- A byte[] secret associated with the token.
- An expiration date after which the token is not valid.
- A String TokenType that categorizes the token.
- An X.509 Certificate associated with the token.
- The principal associated with the token.
- A hashcode that represents the security token (normally the hashcode of the underlying WSS4J object).
- An identifier of another SecurityToken that represents a transformed version of this token.

TokenStores

CXF defines a TokenStore interface for caching SecurityTokens in the WS-Security runtime module. Prior to CXF 2.6, a simple default HashMap based approach was used to cache security tokens. In CXF 2.6, Ehcache is used to provide a suitable default TokenStore implementation to cache security tokens. Tokens are stored until the expiry date of the token if it exists, provided it does not exceed the maximum storage time of 12 hours. If it exceeds this, or if there is no expiry date provided in the security token, it is cached for the default storage time of 1 hour. If the token is expired then it is not cached. This default storage time is configurable. Note that while Ehcache is a compile time dependency of the WS-Security module in CXF, it can be safely excluded in which case CXF will fall back to use the simple HashMap based cache, unless the user specifically wants to implement an alternative TokenStore implementation and configure this instead.

Apache CXF 2.6 provides support for configuring caching via the following JAX-WS properties:

- "org.apache.cxf.ws.security.tokenstore.TokenStore" - The TokenStore instance to use to cache security tokens. By default this uses the EHCacheTokenStore if Ehcache is available. Otherwise it uses the MemoryTokenStore.
- "ws-security.cache.config.file" - Set this property to point to a configuration file for the underlying caching implementation. By default the cxf-ehcache.xml file in the CXF rt-ws-security module is used.

CXF STS token caching

A TokenStore instance can be set on any of the STS operations (see AbstractOperation), and it will be used for caching in the token providers, validators, etc. The STS ships with two TokenStore implementations, a DefaultInMemoryTokenStore which just wraps the EHCacheTokenStore discussed above, and an implementation based on HazelCast.

Token caching works in different ways depending on where it is used. Both token providers (SAML and SecurityContextTokens) that ship with the STS cache a successfully generated token. When a SAML or SecurityContextToken (or UsernameToken) is received by the STS for validation, it will first check to see whether the token is stored in the cache. If it is then validation is skipped. If it is not, then the TokenValidator will re-validate the token, and store it in the cache if validation is successful. A slight caveat to this behaviour for SAML Tokens is that token validation is only skipped if the token that is stored in the cache originally was signed, as this signature value is used to check to see if the two SAML Tokens are equal. The TokenCanceller implementation for cancelling SecurityContextTokens in the STS removes the token from the cache, if the token is stored in the cache and proof-of-possession passes.

Finally, two significant changes in CXF 2.6.0 concerning caching in the STS relate to the length of time tokens are valid for. Prior to CXF 2.6.0, SecurityContextTokens and SAML Tokens issued by the STS were valid for a default (configurable) value of 5 minutes. In CXF 2.6.0, both tokens are now issued for a default value of 30 minutes, and are stored in the cache for this length of time as a result.

Generic Token Handling

This chapter so far has discussed how tokens are provided, validated, and cancelled in the STS. These operations are (at least in theory) relatively independent of WS-Trust. For example, they could be used as an API to provide/validate/process, etc. tokens.

In this section we'll be exploring the larger picture of how this internal token handling functionality works in the context of a client invocation. In this section we will cover some common functionality that is used by all of the WS-Trust operations in the STS implementation.

AbstractOperation

Earlier in this chapter the STS provider framework in Apache CXF was introduced. A number of interfaces were defined for each of the operations that can be invoked on the STS. Before looking at the implementations of these interfaces that ship with the STS, we will look a base class that all of the operations extend, namely the AbstractOperation class. This class defines a number of properties that are shared with any subclasses, and can be accessed via set/get methods:

- STSPropertiesMBean stsProperties - A configuration MBean that holds the configuration for the STS as a whole, such as information about the private key to use to sign issued tokens, etc.
- boolean encryptIssuedToken - Whether to encrypt an issued token or not. The default is false.
- List<ServiceMBean> services - A list of ServiceMBean objects, which correspond to "known" services.
- List<TokenProvider> - A list of TokenProvider implementations to use to issue tokens.
- boolean returnReferences - Whether to return SecurityTokenReference elements to the client or not, that point to the issued token. The default is true.
- TokenStore tokenStore - A cache used to store/retrieve tokens.
- List<TokenValidator> tokenValidators - A list of TokenValidator implementations to use to validate tokens.
- ClaimsManager claimsManager - An object that is used to handle claims.

Several of the properties refer to issuing tokens - this is because this functionality is shared between the issuing and validating operations. At least one TokenProvider implementation must be configured, if the STS is to support issuing a token. Some of these properties have been discussed previously, for example the TokenStore cache covered earlier. This cache could be shared across a number of different operations, or else kept separate. AbstractOperation also contains some common functionality to parse requests, encrypt tokens, create references to return to the client, etc.

STSPropertiesMBean

The AbstractOperation object must be configured with an STSPropertiesMBean object. This is an interface that encapsulates some configuration common to a number of different operations of the STS:

- `void configureProperties()` - load and process the properties
- `void setCallbackHandler(CallbackHandler callbackHandler)` - Set a CallbackHandler object. This is used in the TokenProviders/TokenValidators to retrieve passwords for various purposes.
- `void setSignatureCrypto(Crypto signatureCrypto)` - Set a WSS4J Crypto object to use to sign tokens, or validate signed requests, etc.
- `void setSignatureUsername(String signatureUsername)` - Set the default signature username to use (e.g. corresponding to a keystore alias)

- `void setEncryptionCrypto(Crypto encryptionCrypto)` - Set a WSS4J Crypto object to use to encrypt issued tokens.
- `void setEncryptionUsername(String encryptionUsername)` - Set the default encryption username to use (e.g. corresponding to a keystore alias)
- `void setIssuer(String issuer)` - Set the default issuer name of the STS
- `void setSignatureProperties(SignatureProperties signatureProperties)` - Set the SignatureProperties object corresponding to the STS.
- `void setRealmParser(RealmParser realmParser)` - Set the object used to define what realm a request is in.
- `void setIdentityMapper(IdentityMapper identityMapper)` - Set the object used to map identities across realms.

The STS ships with a single implementation of the STSPropertiesMBean interface - StaticSTSProperties. This class has two additional methods:

- `void setSignaturePropertiesFile(String signaturePropertiesFile)`
- `void setEncryptionPropertiesFile(String encryptionPropertiesFile)`

If no Crypto objects are supplied to StaticSTSProperties, then it will try to locate a properties file using these values, and create a WSS4J Crypto object internally from the properties that are parsed.

SignatureProperties

A SignatureProperties object can be defined on the STSPropertiesMBean. Note that this is unrelated to the signaturePropertiesFile property of StaticSTSProperties. This class provides some configuration relating to the signing of an issued token, as well as symmetric key generation. It has the following properties:

- boolean useKeyValue - Whether to use a KeyValue or not to refer to a certificate in a signature. The default is false.
- long keySize - The (default) key size to use when generating a symmetric key. The default is 256 bits.
- long minimumKeySize - The minimum key size to use when generating a symmetric key. The requestor can specify a KeySize value to use. The default is 128 bits.
- long maximumKeySize - The maximum key size to use when generating a symmetric key. The requestor can specify a KeySize value to use. The default is 512 bits.
- signatureAlgorithm - Signature algorithm preferred by the client. Default value is rsa-sha1
- acceptedSignatureAlgorithms - Alternative signature algorithms that may be used by the STS.
- c14nAlgorithm - Canonicalization algorithm (default c14n-excl-omit-comments) preferred by the client.
- acceptedC14nAlgorithms - Alternative canonicalization algorithms that may be used by the STS.

For example, when the client sends a "KeySize" element to the STS when requesting a SAML Token (and sending a SymmetricKey KeyType URI), the SAMLTokenProvider will check that the requested keysize falls in between the minimum and maximum key sizes defined above. If it does not, then the default key size is used.

Request Parsing

The first thing any of the AbstractOperation implementations do on receiving a request is to call some functionality in AbstractOperation to parse the request. This parsing is done by the RequestParser object, which iterates through the objects of the JAXB RequestSecurityTokenType. The request is parsed into two components, TokenRequirements and KeyRequirements, which are available on the

RequestParser object and are subsequently passed to the desired TokenProvider/TokenValidator/etc objects.

TokenRequirements

The TokenRequirements class holds a set of properties that have been extracted and parsed by RequestParser. These properties loosely relate to the token itself, rather than anything to do with keys. The properties that can be set by RequestParser are:

- String tokenType - The desired TokenType URI. This is required if a token is to be issued.
- Element appliesTo - The AppliesTo element that was received in the request. This normally holds a URL that indicates who the recipient of the issued token will be.
- String context - The context attribute of the request.
- ReceivedToken validateTarget - This object holds the contents of a received "ValidateTarget" element, i.e. a token to validate.
- ReceivedToken onBehalfOf - This object holds the contents of a received "OnBehalfOf" element.
- ReceivedToken actAs - This object holds the contents of a received "ActAs" element.
- ReceivedToken cancelTarget - This object holds the contents of a received "CancelTarget" element, i.e. a token to cancel.
- Lifetime lifetime - The requested lifetime of the issued token. This just holds created and expires Strings, that are parsed from the request.
- RequestClaimCollection claims - A collection of requested claims that are parsed from the request.
- Renewing renewing - Holds the wst:Renewing semantics that were received (if any) as part of the request.

The ReceivedToken class mentioned above parses a received token object, which can be a JAXBElement<?> or a DOM Element. If it is a JAXBElement then it must be either a UsernameToken, SecurityTokenReference, or BinarySecurityToken. If it is a reference to a security token in the security header of the request, then this token is retrieved and set as the ReceivedToken instead.

KeyRequirements

The KeyRequirements class holds a set of properties that have been extracted and parsed by RequestParser. These properties contain everything to do with key handling or creation. The properties that can be set by RequestParser are:

- String authenticationType - An optional authentication type URI. This is currently not used in the STS.
- String keyType - The desired KeyType URI.
- long keySize - The requested KeySize to use when generating symmetric keys.
- String signatureAlgorithm - The requested signature algorithm to use when signing an issued token.
- String encryptionAlgorithm - The requested encryption algorithm to use when encrypting an issued token.
- String c14nAlgorithm - The requested canonicalization algorithm to use when signing an issued token.
- String computedKeyAlgorithm - The computed key algorithm to use when creating a symmetric key.
- String keywrapAlgorithm - The requested KeyWrap algorithm to use when encrypting a symmetric key.

- X509Certificate certificate - A certificate that has been extracted from a "UseKey" element, for use in the SAML case when a PublicKey KeyType URI is specified.
- Entropy entropy - This object holds entropy information extracted from the client request for use in generating a symmetric key. Only BinarySecret elements are currently supported.

SecondaryParameters

RequestParser also supports a "SecondaryParameters" element that might be in the request.

This could be extracted from the WSDL of a service provider that specifies an IssuedToken policy by the client and sent to the STS as part of the RequestSecurityToken request. Only KeySize, TokenType, KeyType and Claims child elements are currently parsed.

The TokenIssueOperation

The TokenIssueOperation is an extension of AbstractOperation that is used to issue tokens. It implements the IssueOperation and IssueSingleOperation interfaces in the STS provider framework.

Recall that AbstractOperation uses the RequestParser to parse a client request into TokenRequirements and KeyRequirements objects. TokenIssueOperation populates a TokenProviderParameters object with values extracted from the TokenRequirements and KeyRequirements objects. A number of different processing steps then occur before a TokenProvider implementation is used to retrieve the desired token, comprising of realm parsing, claims handling, and AppliesTo parsing.

Realm Parsing

We have earlier shown how realms are used with TokenProviders to provide tokens, and also how they work with TokenValidators to validate a given token. However, we did not cover how realms are defined in the first place. Recall that the STSPropertiesMBean configuration object defined on AbstractOperation has a RealmParser property. The RealmParser is an interface which defines a pluggable way of defining a realm for the current request. It has a single method:

- `String parseRealm(WebServiceContext context)` - Return the realm of the current request given a WebServiceContext object.

Therefore if you wish to issue tokens in multiple realms, it is necessary to create an implementation of the RequestParser which will return a realm String given a context object. For example, different realms could be returned based on the endpoint URL or a HTTP parameter. This realm will then get used to select a TokenProvider implementation to use to issue a token of the desired type. It will also be used for token validation in a similar way.

AppliesTo parsing

An AppliesTo element contains an address that refers to the recipient of the issued token. If an AppliesTo element was sent as part of the request then the CXF STS requires that it be explicitly handled. This is done by the list of ServiceMBean objects that can be configured on AbstractOperation. The ServiceMBean interface represents a service, and has the following methods (among others):

- `boolean isAddressInEndpoints(String address)` - Return true if the supplied address corresponds to a known address for this service.
- `void setEndpoints(List<String> endpoints)` - Set the list of endpoint addresses that correspond to this service.

The STS ships with a single implementation of this interface, the StaticService. For the normal use-case of handling an AppliesTo element, the user creates a StaticService object and calls setEndpoints with a set of Strings that correspond to a list of regular expressions that match the allowable set

of token recipients (by address). The TokenIssueOperation will extract the URL address from the EndpointReference child of the received AppliesTo element, and then iterate through the list of ServiceMBean objects and ask each one whether the given address is known to that ServiceMBean object. If an AppliesTo address is received, and no ServiceMBean is configured that can deal with that URL, then an exception is thrown.

The ServiceMBean also defines a number of optional configuration options, such as the default KeyType and TokenType Strings to use for that Service, if the client does not supply them. It also allows the user to set a custom EncryptionProperties object, which defines a set of acceptable encryption algorithms to use to encrypt issued tokens for that service.

Token creation and response

Once the TokenIssuerOperation has processed the client request, it iterates through the list of defined TokenProvider implementations to see if each "can handle" the desired token type in the configured realm (if any). If no TokenProvider is defined, or if no TokenProvider can handle the desired token type, then an exception is thrown. Otherwise, a token is created, and a response object is constructed containing the following items:

- The context attribute (if any was specified).
- The Token Type.
- The requested token (possibly encrypted, depending on configuration).
- A number of references to that token (can be disabled by configuration).
- The received AppliesTo address (if any).
- The RequestedProofToken (if a Computed Key Algorithm was used).
- The Entropy generated by the STS (if any, can be encrypted).
- The lifetime of the generated token.
- The KeySize that was used (if any).

TokenIssueOperation Example

Finally, it's time to look at an example of how to spring-load the STS so that it can issue tokens. This particular example uses a security policy that requires a UsernameToken over the symmetric binding. As the STS is a web service, we first define an endpoint:

```
<jaxws:endpoint id="UTSTS"
   implementor="#utSTSProviderBean"
   address="http://.../SecurityTokenService/UT"
   wsdlLocation=".../ws-trust-1.4-service.wsdl"
   xmlns:ns1="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
   serviceName="ns1:SecurityTokenService"
   endpointName="ns1:UT_Port">
   <jaxws:properties>
      <entry key="security.callback-handler" value="..."/>
      <entry key="security.signature.properties"
         value="stsKeystore.properties"/>
   </jaxws:properties>
</jaxws:endpoint>
```

The jaxws:properties are required to parse the incoming message. The CallbackHandler is used to validate the UsernameToken and provide the password required to access the private key defined in

the signature properties parameter. The "implementor" of the jaxws:endpoint is the SecurityTokenServiceProvider class defined in the STS provider framework:

```
<bean id="utSTSProviderBean"
class="org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider">
   <property name="issueOperation" ref="utIssueDelegate"/>
   ...
</bean>
```

This bean supports the Issue Operation via a TokenIssueOperation instance:

```
<bean id="utIssueDelegate"
   class="org.apache.cxf.sts.operation.TokenIssueOperation">
   <property name="tokenProviders" ref="utSamlTokenProvider"/>
   <property name="services" ref="utService"/>
   <property name="stsProperties" ref="utSTSProperties"/>
</bean>
```

This TokenIssueOperation instance has a single TokenProvider configured to issue SAML Tokens (with a default Subject and Attribute statement):

```
<bean id="utSamlTokenProvider"
   class="org.apache.cxf.sts.token.provider.SAMLTokenProvider">
</bean>
```

The TokenIssueOperation also refers to a single StaticService implementation, which in turn defines a single URL expression to use to compare any received AppliesTo addresses:

```
<bean id="utService"
   class="org.apache.cxf.sts.service.StaticService">
   <property name="endpoints" ref="utEndpoints"/>
</bean>
```

```
<util:list id="utEndpoints">
   <value>http://localhost:(\d)*/(doubleit|metrowsp)/services/doubleit   //
      (UT|.*symmetric.*|.*)</value>
</util:list>
```

Finally, the TokenIssueOperation is configured with a StaticSTSProperties object. This class contains properties that define what private key to use to sign issued SAML tokens, as well as the Issuer name to use in the generated token.

```
<bean id="utSTSProperties"
   class="org.apache.cxf.sts.StaticSTSProperties">
   <property name="signaturePropertiesFile" value="stsKeystore.properties"/>
   <property name="signatureUsername" value="mystskey"/>
   <property name="callbackHandlerClass" value="..."/>
   <property name="issuer" value="DoubleItSTSIssuer"/>
   ...
</bean>
```

The TokenCancelOperation

The TokenCancelOperation class is used to cancel tokens in the STS. It implements the CancelOperation interface in the STS provider framework. In addition to the properties that it inherits from AbstractOperation, it has a single property that can be configured:

- List<TokenCanceller> tokencancellers - A list of TokenCanceller implementations to use to cancel tokens.

Recall that AbstractOperation uses the RequestParser to parse a client request into TokenRequirements and KeyRequirements objects. TokenCancelOperation first checks that a "CancelTarget" token was

received and successfully parsed (if so it will be stored in the TokenRequirements object). If no token was received then an exception is thrown.

The TokenCancelOperation then populates a TokenCancellerParameters object with values extracted from the TokenRequirements and KeyRequirements objects. It iterates through the list of defined TokenCanceller implementations to see if any "can handle" the received token. If no TokenCancelle r is defined, or if no TokenCanceller can handle the received token, then an exception is thrown. Otherwise, the received token is cancelled. If there is an error in cancelling the token, then an exception is also thrown. A response is constructed with the context attribute (if applicable), and the cancelled token type.

Claims Handling in the STS

A typical scenario for WS-Trust is when the client requires a particular security token from an STS to access a service provider. The service provider can let the client know what the requirements are for the security token in an IssuedToken policy embedded in the WSDL of the service. In particular, the service provider can advertise the claims that the security token must contain in the policy (either directly as a child element of IssuedToken, or else as part of the RequestSecurityTokenTemplate). An example is contained in the STS systests:

```
<sp:RequestSecurityTokenTemplate>
   <t:TokenType>http://...#SAMLV1.1</t:TokenType>
   <t:KeyType>http://.../PublicKey</t:KeyType>
   <t:Claims Dialect="http://.../identity">
      <ic:ClaimType Uri="http://.../claims/role"/>
   </t:Claims>
</sp:RequestSecurityTokenTemplate>
```

This template specifies that a SAML 1.1 Assertion is required with an embedded X509 Certificate in the subject of the Assertion. The issued Assertion must also contain a "role" claim. The template is sent verbatim by the client to the STS when requesting a security token.

Parsing claims

The RequestParser object parses the client request into TokenRequirements and KeyRequirement s objects. As part of this processing it converts a received Claims element into a RequestClaimCo llection object. The RequestClaimCollection is just a list of RequestClaim objects, along with a dialect URI. The RequestClaim object holds the claimType URI as well as a boolean indicating whether the claim is optional or not.

The ClaimsHandler

The ClaimsHandler is an interface that the user must implement to be able to "handle" a requested claim. It has two methods:

- `List<URI> getSupportedClaimTypes()` - Return the list of ClaimType URIs that this ClaimHandler object can handle.
- `ClaimCollection retrieveClaimValues (Principal p, RequestClaimCo llection rcc)` - Return the claim values associated with the requested claims (and client principal).

The ClaimCollection object that is returned is just a list of Claim objects. This object represents a Claim that has been processed by a ClaimsHandler instance. It essentially contains a number of properties that the ClaimsHandler implementation will set, e.g.:

- URI claimType - The claimtype URI as received from the client.
- String value - The claim value

Each Claim object in a ClaimCollection corresponds to a RequestClaim object in the RequestClaimCollection, and contains the Claim value corresponding to the requested claim. The STS ships with a single ClaimsHandler implementation, the LDAPClaimsHandler, which can retrieve claims from an LDAP store. A simpler example is available in the unit tests.

The ClaimsManager

The ClaimsManager defined on AbstractOperation holds a list of ClaimsHandler objects. So to support claim handling in the STS, it is necessary to implement one or more ClaimsHandler objects for whatever Claim URIs you wish to support, and register them with a ClaimsManager instance, which will be configured on the TokenIssueOperation object.

As detailed in the previous article, the TokenIssueOperation gets the realm of the current request, and does some processing of the AppliesTo address, after the RequestParser has finished parsing the request. The RequestClaimCollection object that has been constructed by the RequestParser is then processed. For each RequestClaim in the collection, it checks to see whether the ClaimsManager has a ClaimsHandler implementation registered that can "handle" that Claim (by checking the URIs). If it does not, and if the requested claim is not optional, then an exception is thrown.

If a ClaimsHandler implementation is registered with the ClaimsManager that can handle the desired claim, then the claims are passed through to the TokenProvider implementation, which is expected to be able to invoke the relevant ClaimHandler object, and insert the processed Claim into the generated security token. How this is done is entirely up to the user. For example, for the use-case given above of a SAML 1.1 token containing a "role" claim, the user could implement a custom AttributeStatementProvider instance that evaluates the claim values (via a custom ClaimsHandler implementation registered with the ClaimsManager) and constructs a set of Attributes accordingly in an AttributeStatement. An example of how to do this is given in the CXF unit tests.

The TokenValidateOperation

TokenValidateOperation is an extension of AbstractOperation used to validate tokens in the STS. It implements the ValidateOperation interface in the STS provider framework. For validation, the below property from AbstractOperation can be configured:

- List<TokenValidator> tokenValidators - A list of TokenValidator implementations to use to validate tokens.

Recall that AbstractOperation uses the RequestParser to parse a client request into TokenRequirements and KeyRequirements objects. TokenValidateOperation first checks that a "ValidateTarget" token was received and successfully parsed (if so it will be stored in the TokenRequirements object). If no token was received then an exception is thrown.

Token validation and response

TokenValidateOperation then populates a TokenValidatorParameters object with values extracted from the TokenRequirements and KeyRequirements objects. It iterates through the list of defined TokenValidator implementations to see if any "can handle" the received token. If no TokenValidator is defined, or if no TokenValidator can handle the received token, then an exception is thrown. Otherwise, the received token is validated. The TokenValidateOperation then checks to see whether token transformation is required.

Token Transformation

If the received token is successfully validated, and if the client supplies a TokenType in the request that does not correspond to the WS-Trust "status" namespace, then the TokenValidateOperation attempts to transform the validated token into a token of the requested type. Token transformation works in a similar way to token issuing, as detailed previously. A TokenProviderParameters object is constructed and the same processing steps (Realm parsing, AppliesTo parsing) are followed as for token issuing.

One additional processing step occurs before the token is transformed. If the TokenValidatorResponse object has a principal that was set by the TokenValidator implementation, then it is set as the principal of the TokenProviderParameters object. However, it is possible that the token is being issued in a different realm to that of the validated token, and the principal might also need to be transformed. Recall that the STSPropertiesMBean configuration object defined on AbstractOperation has an IdentityMapper property. This interface is used to map identities across realms. It has a single method:

- `Principal mapPrincipal(String sourceRealm, Principal sourcePrincipal, String targetRealm)` - Map a principal from a source realm to a target realm

If the source realm is not null (the realm of the validated token as returned in TokenValidator Response), and if it does not equal the target realm (as set by the RealmParser), then the identity mapper is used to map the principal to the target realm and this is stored in TokenProviderParameters for use in token generation. After the (optional) identity mapping step, TokenValidateOperation iterates through the TokenProvider list to find an implementation that can "handle" the desired token type in the given (target) realm (if applicable). If no TokenProvider is defined, or if no TokenProvider can handle the desired token type, then an exception is thrown.

Token response

After token validation has been performed, and after any optional token transformation steps, a response object is constructed containing the following items:

- The context attribute (if any was specified).
- The received Token Type (if any was specified, or the "status" token type if validation was successful).
- Whether the received token was valid or not (status code & reason).
- If the received token was valid, and if token transformation successfully occurred:
    - The transformed token.
    - The lifetime of the transformed token.
    - A number of references to that token (can be disabled by configuration).

TokenValidateOperation example

Finally, it's time to look at an example of how to spring-load the STS so that it can validate tokens. This particular example uses a security policy that requires a UsernameToken over the transport binding (client auth is disabled). As the STS is a web service, we first define an endpoint:

```
<jaxws:endpoint id="transportSTS"
   implementor="#transportSTSProviderBean"
   address="http://.../SecurityTokenService/Transport"
   wsdlLocation".../ws-trust-1.4-service.wsdl"
   xmlns:ns1="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
   serviceName="ns1:SecurityTokenService"
   endpointName="ns1:Transport_Port">
   <jaxws:properties>
      <entry key="security.callback-handler" value="..."/>
   </jaxws:properties>

</jaxws:endpoint>
```

The CallbackHandler JAX-WS property is used to validate the UsernameToken. The "implementor" of the jaxws:endpoint is the SecurityTokenServiceProvider class defined in the STS provider framework:

```
<bean id="transportSTSProviderBean"
   class="org.apache.cxf.ws.security.sts.provider.  //
      SecurityTokenServiceProvider">
   ...
   <property name="validateOperation" ref="transportValidateDelegate"/>
</bean>
```

This bean supports the Validate Operation via a TokenValidateOperation instance:

```
<bean id="transportValidateDelegate"
   class="org.apache.cxf.sts.operation.TokenValidateOperation">
   <property name="tokenValidators" ref="transportTokenValidators"/>
   <property name="stsProperties" ref="transportSTSProperties"/>
</bean>
```

This TokenValidateOperation instance has a number of different TokenValidator instances configured:

```
<util:list id="transportTokenValidators">
   <ref bean="transportSamlTokenValidator"/>
   <ref bean="transportX509TokenValidator"/>
   <ref bean="transportUsernameTokenValidator"/>
</util:list>

<bean id="transportX509TokenValidator"
   class="org.apache.cxf.sts.token.validator.X509TokenValidator"/>

<bean id="transportUsernameTokenValidator"
   class="org.apache.cxf.sts.token.validator.UsernameTokenValidator"/>

<bean id="transportSamlTokenValidator"
   class="org.apache.cxf.sts.token.validator.SAMLTokenValidator"/>
</bean>
```

Finally the STSPropertiesMBean object that is used is given as follows:

```
<bean id="transportSTSProperties"
   class="org.apache.cxf.sts.StaticSTSProperties">
   <property name="signaturePropertiesFile" value="..."/>
   <property name="signatureUsername" value="mystskey"/>

   <property name="callbackHandlerClass" value="..."/>
   <property name="encryptionPropertiesFile" value="..."/>
   <property name="issuer" value="DoubleItSTSIssuer"/>
   <property name="encryptionUsername" value="myservicekey"/>
</bean>
```

DefaultSecurityTokenServiceProvider

The CXF STS provides a subclass of the SecurityTokenServiceProvider to efficiently handle the most common use cases when configuring an STS. This implementation supports the issue and validate bindings by default, supports the ability to issue and validate SAML Tokens, and validates UsernameTokens and X.509 Tokens. Therefore, for any of these use cases there is no need to use the SecurityTokenServiceProvider directly, which involves explicitly instantiating each TokenProvider or TokenValidator instance and setting them on an IssueOperation/ValidateOperation instance. However, if you wish to use the renew or cancel bindings, or do anything with SecurityContextTokens, configuring SecurityTokenServiceProvider will be necessary.

The DefaultSecurityTokenServiceProvider can be configured with the following optional properties:

- boolean encryptIssuedToken - Whether to encrypt the issued token or not. The default is false.
- List<ServiceMBean> services - A list of service endpoints to support.
- boolean returnReferences - Whether to return references to an issued token or not. The default is true.
- TokenStore tokenStore - The TokenStore caching implementation to use.
- ClaimsManager claimsManager - The ClaimsManager to use if you wish to be able to handle claims.

A sample CXF configuration file showing the standard SecurityTokenServiceProvider is here, and see here for a simplified configuration using DefaultSecurityTokenServiceProvider.

Using STS with the Talend Runtime

This chapter describes the deployment and configuration of STS with a Talend Runtime Container, how to configure the Data Services to use the STS. It also discusses creating keys and certificates for STS and clients.

Note: The term <TalendRuntimePath> is used for the directory where Talend Runtime is installed. This is typically the full path of either Runtime_ESBSE or Talend-ESB-V, depending on the version of the software that is being used. Please substitute appropriately.

Deploying the STS into the Talend Runtime Container

About this task

Warning: For production use, the sample keys used here will need to be replaced with your project's own keys, usually signed by a third-party CA.

To enable Security Token Service in the Talend Runtime, you need to deploy it into a Talend Runtime Container:

Procedure

1. Replace the STS' sample keystore/truststore called `stsstore.jks` located in the `<TalendRuntimePath>/container/etc/keystores` folder with your own keystore. See Security Token Service Configuration on page 37 for more information.
2. cd `<TalendRuntimePath>/container/bin` directory, enter `trun` to start Talend Runtime, a Talend Runtime Container(Karaf) console window will open.
3. In the console, type `tesb:start-sts` to install the Security Token Service feature. Or type `feature:install tesb-sts` if you are using a generic Karaf container instead of Talend Runtime
4. Type `list | grep STS` in the console. You should see the following output:

```
ID      State      Blueprint  Spring     Level  Name
[ 203] [Active ]   [        ] [started ] [ 60]  Apache CXF STS Core
 (2.5.0)
Fragments: 204
[ 204] [Resolved ] [        ] [        ] [ 60]  Talend :: ESB ::
 STS :: CONFIG ()
```

The above shows that the Security Token Service feature is enabled in the Talend Runtime Container. The Fragment Bundle `204: Talend :: ESB :: STS :: CONFIG ()` provides the custom configuration about the Security Token Service, which will be described in Security Token Service Configuration on page 37.

Deploying the STS into a Servlet Container (Tomcat)

About this task

Warning: For production use, the sample keys used here will need to be replaced with your project's own keys, usually signed by a third-party CA.

To enable Security Token Service using a servlet container (here Tomcat is used as an example), follow the below steps:

Procedure

1. Extract the `<TalendRuntimePath>/add-ons/sts/SecurityTokenService.war` file and replace the `stsstore.jks` STS sample keystore/truststore with your own keystore. Alter the `stsKeystore.properties` file with any different configuration information based on your new keystore. Recompress the extracted WAR into a new WAR file.
2. Deploy the new WAR file created in the previous step into the Tomcat container.
3. Start Tomcat and open a browser with the follow url: `http://{tomcat}host:port/SecurityTokenService/`. You will see several Security Token Service available, such as Username Token service (UT), X.509 Token service, and so on.
4. Enter URL: `http://{tomcat host}:port/SecurityTokenService/UT?wsdl`, the displayed WSDL file will describe the details about the Security Token Service.

Security Token Service Configuration

The Security Token Service provides the following methods as described in the below snippet, which is defined in `SecurityTokenService.war/WEB-INF/wsdl/ws-trust-1.4-service.wsdl`

```
<wsdl:service name="SecurityTokenService">
   <wsdl:port name="UT_Port" binding="tns:UT_Binding">
      <soap:address location=
         "http://localhost:8080/SecurityTokenService/UT"/>
   </wsdl:port>
   <wsdl:port name="X509_Port" binding="tns:X509_Binding">
      <soap:address location=
         "http://localhost:8080/SecurityTokenService/X509"/>
   </wsdl:port>
   <wsdl:port name="Transport_Port" binding="tns:Transport_Binding">
      <soap:address location="/Transport"/>
   </wsdl:port>
   <wsdl:port name="UTEncrypted_Port" binding="tns:UTEncrypted_Binding">
      <soap:address location="/UTEncrypted"/>
   </wsdl:port>
</wsdl:service>
```

As above snippet shows, the Security Token Service can issue (or validate) UserName Token or X509 Token, and so on.

In Talend Runtime Container, the configuration of Security Token Service can be defined in the file `<TalendRuntimePath>/etc/org.talend.esb.sts.server.cfg`:

```
stsServiceUrl=/SecurityTokenService/UT
stsX509ServiceUrl=/SecurityTokenService/X509
loginModule=TIDM
jaasContext=karaf
signatureProperties=file:${tesb.home}/etc/keystores/stsKeystore.properties
signatureUsername=mystskey
bspCompliant=false
tidmServiceUrl=http://localhost:8080/syncope/cxf/
tidmUsername=admin
tidmPassword=password
useMessageLogging=false
samlTokenLifetime=1800
```

By default, Security Token Service is configured to use JAAS interface to verify the user credentials and perform authentication. As shown above, Security Token Service uses `karaf` JAAS Context which is the default context configured for Talend Runtime Container and uses `PropertiesLoginModule` of Karaf. This login module uses the file located in `/etc/users.properties` which contains a list of users and their password, hence the users which are needed to be authenticated via the Security Token Service should be listed here. A different login module can be configured for the Security Token Service by updating the `jaasContext` parameter in the above configuration. A Talend Runtime Container comes with several login modules that can be used to integrate into your environment, the modules are listed below:

- PropertiesLoginModule
- OsgiConfigLoginModule
- JDBCLoginModule
- LDAPLoginModule

The `signatureProperties` file, which is located in: `/etc/keystores/stsKeystore.properties`, defines the signature configuration as shown below:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.alias=mystskey
org.apache.ws.security.crypto.merlin.keystore.file=./etc/keystores/stsstore.jks
```

The keystore file name can be changed by altering its value in the `stsKeystore.properties` file. With the default configuration as shown above, the Talend Runtime Container will expect the STS' private key to have the alias of `mystskey`, this can be changed by altering the `alias` and `signatureUsername` values in the two configuration files listed above.

The `samlTokenLifetime` property in the `<TalendRuntimePath>/etc/org.talend.esb.sts.server.cfg` file allows you to set the lifetime of the SAML token. The default is 1800 seconds. You can change it as needed.

Setting up the security management system in Security Token Service

The Security Token Service is provided with all versions of Talend ESB, however the security management system behind it is different in the community and in the subscription version. For the community version, Talend Open Studio for ESB, the security service is managed via the JAAS authentication handler, whereas for the subscription version, Talend ESB, the security service is, by

default, managed by the Talend Identity and Access Management, based on Apache Syncope. The use of the JAAS is also possible, within Talend ESB, by switching the module used from Talend Identity and Access Management to JAAS.

So, if you are using the subscription version of Talend ESB, you are able to either use the Talend Identity and Access Management or the JAAS security management systems. To switch between those two systems, you have to change the `loginModule` value in the `<TalendRuntimePath>/container/etc/org.talend.esb.sts.server.cfg` configuration file:

- To use Talend Identity and Access Management, set the parameter as follows: `loginModule=TIDM`. You also need to set the `tidmServiceUrl`, `tidmUsername`, and `tidmPassword` properties in the configuration file. For more information about how to install the Talend Identity and Access Management, see the *Talend Installation and Upgrade Guide* .
- To use JAAS, set the parameter as follows: `loginModule=JAAS`.

This way, when executing the `tesb:start-sts` command, Talend Runtime Container checks which module is used and then installs either the **tesb-sts** or the **tesb-sts-tidm** feature. If the `loginModule` property does not exist, by default, **tesb-sts-tidm** will be installed.

To switch from one security management system to the other, simply use the following commands:

- ```
  tesb:switch-sts-jaas
  ```

  If the Security Token Service is not started yet, this command only changes the configuration file property to `loginModule=JAAS`.

  If the Security Token Service using the Talend Identity and Access Management is started, this command stops it and starts the JAAS module instead.
- ```
  tesb:switch-sts-tidm
  ```

  If the Security Token Service is not started yet, this command only changes the configuration file property to `loginModule=TIDM`.

  If the Security Token Service using the JAAS module is started, this command stops it and starts the Talend Identity and Access Management instead.

Setting up logging parameters in Security Token Service

If you are using Talend Identity and Access Management with STS to manage authorization accesses to services in Talend ESB(only available in Talend subscription products), you can decide whether to log the communication between the modules involved.

To do so, you have to edit the following configuration files:

- In `<TalendRuntimePath>/container/etc/org.talend.esb.sts.server.cfg`, set `useMessageLogging=true` to indicate whether the communication between STS and Talend Identity and Access Management should be logged. Dynamic reconfiguration at runtime is supported.

  By default, the option is disable: `useMessageLogging=false`.
- In `<TalendRuntimePath>/container/etc/org.talend.esb.authorization.pdp.cfg`, set `useMessageLogging=true` to indicate whether the communication between PDP and the Talend Authorization XACML Repository should be logged. Dynamic reconfiguration at runtime is supported.

By default, the option is disable: `useMessageLogging=false`.

Data Service Configuration for using STS

In the Talend Runtime Container, the configuration used by Data Service Consumers for using Security Token Service (STS) can be defined in the file: `<TalendRuntimePath>/container/etc/org.talend.esb.job.client.sts.cfg`.

```
#STS endpoint configuration
sts.wsdl.location = \
   http://localhost:8040/services/SecurityTokenService/UT?wsdl
sts.namespace = http://docs.oasis-open.org/ws-sx/ws-trust/200512/
sts.service.name = SecurityTokenService
sts.endpoint.name = UT_Port

#STS properties configuration
security.sts.token.username = myclientkey
security.sts.token.usecert = true
ws-security.is-bsp-compliant = false
security.sts.token.properties = \
   file:${tesb.home}/etc/keystores/clientKeystore.properties
```

The STS endpoint used by the consumer is defined by `sts.wsdl.location`. This configuration should be changed in case the STS service is running on a different host and port. The keystore configuration described above is used for signing the timestamp sent in the request by the consumer to the provider. The Talend ESB-supplied sample keystores and certificates above are not meant for production use. Be sure to use your own keys (with different passwords) and configure them as discussed below.

A Data Service consumer can use two types of authentication mechanisms: **Username token** and **SAML token**.

- When using **Username token**, the consumer sends the credentials as a part of the request to the provider and authentication is performed on the provider side. The policy used by the consumer for Username token authentication is defined in the file `<TalendRuntimePath>/etc/org.talend.esb.job.token.policy`.
- For **SAML token**, the consumer makes a SAML token issue request to the STS passing its credentials and on successful authentication the STS issues a SAML token. This SAML token is sent as a part of the request to the provider and the provider verifies the validity of the SAML token. The policy used by the consumer for SAML token authentication is defined in the file `<TalendRuntimePath>/etc/org.talend.esb.job.saml.policy`.

When using **Username token**, a Data Service provider receives credentials from the consumer and performs authentication locally. By default a Data Service provider is configured with JAAS authentication handler and uses the default JAAS context `karaf` configured for the Talend Runtime Container. The login module configured for this context uses the file located in `/etc/users.properties`, which contains a list of users and their password. Thus, the user which needs to be authenticated should be listed here.

In the case of a SAML token, the provider locally verifies the integrity of the token using a certificate, the configuration for it is defined in the file `<TalendRuntimePath>/etc/org.talend.esb.job.service.cfg`.

```
security.signature.properties = \
   file:${tesb.home}/etc/keystores/serviceKeystore.properties
security.signature.username = myservicekey
security.signature.password = skpass
```

Creating keys for the Security Token Service

This section describes how to create keys for the Security Token Service. We highly recommend that you use third-party signed CA's (certificate authorities) or create your own Certificate Authority, but the following instructions can be used to create self-signed keys.

Using OpenSSL to create certificates

First, create the keys.

Warning: Replace "<PW-Sk>", "<PW-Sk>","<PW-Cs>" and "<PW-Ck>" in the example below with your own passwords.

Creating the service keystore

Note: given the `rm` commands below, it is probably best to create a new directory and navigate to it before running these commands from a terminal window.

```
rm *.p12 *.pem *.jks *.cer
openssl req -x509 -days 3650 -newkey rsa:1024 -keyout servicekey.pem -out
    servicecert.pem -passout pass:<PW-Sk>
```

When running this `openssl` command, enter any geographic and company information desired, the key password in passout, and a common name of your choice (perhaps `servicecn` for the service and `clientcn` for the client).

```
openssl pkcs12 -export -inkey servicekey.pem -in servicecert.pem -out
   service.p12 -name myservicekey -passin pass:<PW-Sk> -passout
   pass:<PW-Sk>
```

This creates a pkcs12 certificate. Note the <PW-Sk> value will be used both for the keystore and the private key itself.

```
keytool -importkeystore -destkeystore servicestore.jks -deststorepass
   <PW-Sk> -deststoretype jks -srckeystore service.p12 -srcstorepass
   <PW-Sk> -srcstoretype pkcs12 # See Note 3
```

This places the certificate in a new JKS keystore. The keystore's password is changed here to <PW-Sk>, but the private key's password retains the earlier value of <PW-Sk>.

```
keytool -list -keystore servicestore.jks -storepass <PW-Sk> -v
```

The list command is just to show the keys presently in the keystore.

```
keytool -exportcert -alias myservicekey -storepass <PW-Sk> -keystore
   servicestore.jks -file service.cer
keytool -printcert -file service.cer
rm *.pem *.p12
```

Creating the client keystore

```
openssl req -x509 -days 3650 -newkey rsa:1024 -keyout clientkey.pem
   -out clientcert.pem -passout pass:<PW-Cs>
openssl pkcs12 -export -inkey clientkey.pem -in clientcert.pem
   -out client.p12
   -name myclientkey -passin pass:<PW-Cs> -passout pass: <PW-Ck>
keytool -importkeystore -destkeystore clientstore.jks -deststorepass
   <PW-Cs> -deststoretype jks -srckeystore client.p12
   -srcstorepass <PW-Ck>-srcstoretype pkcs12
keytool -list -keystore clientstore.jks -storepass <PW-Cs> -v
keytool -exportcert -alias myclientkey -storepass <PW-Cs> -keystore
   clientstore.jks -file client.cer
keytool -printcert -file client.cer
rm *.pem *.p12
```

Deploying and Using a Security Token Service (STS)

You have created the service and client keystores as in the previous section. Now create the STS keystore as follows:

Note: Replace <PW-Ts>, <PW-Tk> in the example below with your own passwords.

```
openssl req -x509 -days 3650 -newkey rsa:1024 -keyout stskey.pem -out
   stscert.pem -passout pass:<PW-Ts>
openssl pkcs12 -export -inkey stskey.pem -in stscert.pem -out sts.p12
   -name mystskey -passin pass:<PW-Ts> -passout pass:<PW-Tk>
keytool -importkeystore -destkeystore stsstore.jks -deststorepass <PW-Ts>
   -srckeystore sts.p12 -srcstorepass <PW-Tk> -srcstoretype pkcs12
keytool -list -keystore stsstore.jks -storepass <PW-Ts>
keytool -exportcert -alias mystskey -storepass <PW-Ts> -keystore
   stsstore.jks -file sts.cer
keytool -printcert -file sts.cer
rm *.pem *.p12
```

To fix any issues with fixed paths to the keystore and truststore locations within the WSDLs, the source code download uses Maven resource filtering to allow for a relative path to the project base directory to be used instead.

Next, the service keystore will need to have the STS public key added so it trusts it, and vice-versa. Also, the client will need to have the STS' and WSP's certificates added to its truststore, as it relies on symmetric binding to encrypt the SOAP requests it makes to both:

```
keytool -keystore servicestore.jks -storepass <PW-Sk> -import -noprompt
   -trustcacerts -alias mystskey -file sts.cer
keytool -keystore stsstore.jks -storepass <PW-Ts> -import -noprompt
   -trustcacerts -alias myservicekey -file service.cer
keytool -keystore clientstore.jks -storepass <PW-Cs> -import -noprompt
   -trustcacerts -alias mystskey -file sts.cer
keytool -keystore clientstore.jks -storepass <PW-Cs> -import -noprompt
   -trustcacerts -alias myservicekey -file service.cer
```

If you plan on using X.509 authentication of the WSC to the STS (instead of UsernameToken), the former's public key will need to be in the latter's truststore. This can be done with the following commands:

```
keytool -exportcert -alias myclientkey -storepass <PW-Cs> -keystore
   clientstore.jks -file client.cer
keytool -keystore stsstore.jks -storepass <PW-Ts> -import -noprompt
   -trustcacerts -alias myclientkey -file client.cer
```

Since the service does not directly trust the client (the purpose for our use of the STS to begin with), we will not add the client's public certificate to the service's truststore as normally done with message-layer encryption.

Secure Token Service (STS) Client Configuration

Now we look at the configuration needed on the client side in order to interact with the STS.

- First we look at a simple example in the CXF's basic STS system test "IssueUnitTest".
- Then we run the JAX-WS CXF WS-Trust sample that ships with Talend ESB.

STS Client Behavior

A simple example of how a CXF client can obtain a security token from the STS is shown in the CXF's basic STS system test "IssueUnitTest". This test starts an instance of the new CXF STS and obtains a number of different security tokens, all done completely programmatically, i.e. with no Spring configuration. The STS instance that is used for the test cases is configured with a number of different endpoints that use different security bindings (defined in the wsdl of the STS). For the purposes of this test, the Transport binding is used:

```
<wsp:Policy wsu:Id="Transport_policy">
   <wsp:ExactlyOne>
      <wsp:All>
         <sp:TransportBinding
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
               <sp:TransportToken>
                  <wsp:Policy>
                     <sp:HttpsToken RequireClientCertificate="false"/>
                  </wsp:Policy>
               </sp:TransportToken>
               <sp:AlgorithmSuite>
                  <wsp:Policy>
                     <sp:Basic128 />
                  </wsp:Policy>
               </sp:AlgorithmSuite>
               <sp:Layout>
                  <wsp:Policy>
                     <sp:Lax />
                  </wsp:Policy>
               </sp:Layout>
               <sp:IncludeTimestamp />
            </wsp:Policy>
         </sp:TransportBinding>
         <sp:SignedSupportingTokens
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
               <sp:UsernameToken
                  sp:IncludeToken=".../AlwaysToRecipient">
                  <wsp:Policy>
                     <sp:WssUsernameToken10 />
                  </wsp:Policy>
               </sp:UsernameToken>
            </wsp:Policy>
         </sp:SignedSupportingTokens>
         ...
      </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

In other words, this security policy requires that a one-way TLS connection must be used to communicate with the STS, and that authentication is performed via a Username Token in the SOAP header.

The object that communicates with an STS in CXF is the STSClient. Typically, the user constructs an STSClient instance (normally via Spring), sets it with certain properties such as the WSDL location of the STS, what service/port to use, various crypto properties, etc, and then stores this object on the

message context using the SecurityConstants tag "security.sts.client". This object is then controlled by the IssuedTokenInterceptorProvider in the ws-security runtime in CXF. This interceptor provider is triggered by the "IssuedToken" policy assertion, which is typically in the WSDL of the service provider. This policy assertion informs the client that it must obtain a particular security token from an STS and include it in the service request. The IssuedTokenInterceptorProvider takes care of using the STSClient to get a Security Token from the STS, and handles how long the security token should be cached, etc.

An example of a simple IssuedToken policy that might appear in the WSDL of a service provider is as follows:

```
<sp:IssuedToken sp:IncludeToken=".../AlwaysToRecipient">
   <sp:RequestSecurityTokenTemplate>
      <t:TokenType>
         http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0
      </t:TokenType>
      <t:KeyType>
         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer
      </t:KeyType>
   </sp:RequestSecurityTokenTemplate>
   ...
</sp:IssuedToken>
```

This policy states that the client should include a SAML 2.0 Assertion of subject confirmation method "Bearer" in the request. The client must know how to communicate with an STS to obtain such a token. This is done by providing the STSClient object with the appropriate information.

The IssueUnitTest referred to above uses the STSClient programmatically to obtain a security token. Let's look at the "requestSecurityToken" method called by the tests. An STSClient is instantiated via the CXF bus, and the WSDL location of the STS, plus service and port names are configured:

```
STSClient stsClient = new STSClient(bus);
stsClient.setWsdlLocation("https://.../SecurityTokenService/Transport?wsdl");
stsClient.setServiceName("{...}SecurityTokenService");
stsClient.setEndpointName("{...}Transport_Port");
```

A map is then populated with various properties and set on the STSClient. It is keyed with a different number of SecurityConstants tags. A username is supplied for use as the "user" in the UsernameToken. A CallbackHandler class is supplied to get the password to use in the UsernameToken. Compliance of the Basic Security Profile 1.1 is turned off, this is to prevent CXF throwing an exception when receiving a non-spec compliant response from a non-CXF STS:

```
Map<String, Object> properties = new HashMap<String, Object>();
stsClient.setProperties(properties);
properties.put(SecurityConstants.USERNAME, "alice");
properties.put(
   SecurityConstants.CALLBACK_HANDLER,
   "org.apache.cxf.systest.sts.common.CommonCallbackHandler"
);
properties.put(SecurityConstants.IS_BSP_COMPLIANT, "false");
```

If the KeyType is a "PublicKey", then an X.509 Certificate is presented to the STS in the request to embed in the generated SAML Assertion. The X.509 Certificate is obtained from the keystore defined in "clientKeystore.properties", with the alias "myclientkey". Finally, the "useCertificateForConfirmat

ionKeyInfo" property of the STSClient means that the entire certificate is to be included in the request, instead of a KeyValue (which is the default):

```
if (PUBLIC_KEY_KEYTYPE.equals(keyType)) {
   properties.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
   properties.put(SecurityConstants.STS_TOKEN_PROPERTIES,
      "clientKeystore.properties");
   stsClient.setUseCertificateForConfirmationKeyInfo(true);
}
```

Finally, the token type is set on the STSClient (the type of token that is being requested), as well as the KeyType (specific to a SAML Assertion), and a security token is requested, passing the endpoint address which is sent to the STS in the "AppliesTo" element:

```
stsClient.setTokenType(tokenType);
stsClient.setKeyType(keyType);
return stsClient.requestSecurityToken(endpointAddress);
```

The returned SecurityToken object contains the received token as a DOM element, the ID of the received token, any reference elements that were returned - which show how to reference the token, any secret associated with the token, and the lifetime of the token.

Running the JAX-WS CXF WS-Trust Sample from Talend ESB

Talend ESB includes a `jaxws-cxf-sts` sample under the examples folder of the distribution. The STS and WSP portions of this example run on Apache Tomcat Version 7.

Note: If you would like to use Tomcat 6, as discussed in this sample's README file, some changes to this sample are needed:

- In Tomcat 6's tomcat-users.xml file (discussed in the next section), instead of giving the tomcat user the "manager-script" and "manager-gui" roles, give him the Tomcat 6-specific "manager" role.
- Use the `-PTomcat6` setting when deploying either the STS or the Web Service Provider onto Tomcat

If not already done, configure Maven to be able to install and uninstall the WSP and the STS by following these instructions:

Download Tomcat and configure Tomcat-Maven integration

Procedure

1. Download from http://tomcat.apache.org/download-70.cgi or http://tomcat.apache.org/download-60.cgi the latest release version of Tomcat and extract the zip or tar.gz file into a new directory.
2. Have an environment variable $CATALINA_HOME point to your expanded Tomcat application directory, e.g. for Linux (in your ~/.bashrc file): `export CATALINA_HOME=/usr/myapps/tomcat-<version>`
3. In the `CATALINA_HOME/conf/tomcat-users.xml` file, we'll need to create a user with appropriate manager permissions. Create a new user with the role of manager-script or give the default "tomcat" user the manager-script role as shown below. This role allows for deploying web

applications using scripting tools such as the Tomcat Maven Plugin we're using in this tutorial. Although not necessary for Tomcat deployment, The manager-gui role gives ability to access the browser-based Tomcat Manager HTML application, helpful for a quick authentication check. Depending on your security needs, you may or may not wish to do this in production.

```
<tomcat-users>
   ...other entries...
   <role rolename="manager-script"/>
   <role rolename="manager-gui"/>
   <user username="tomcat" password="????"
      roles="tomcat,manager-script,manager-gui"/>
</tomcat-users>
```

Warning: For production it is best to grant manager roles to another username besides the easy-to-guess default "tomcat" user.

Change the "????" in user password line of tomcat-users configuration to another appropriate password and save.

4. Start Tomcat from a console window: `{prompt}% $CATALINA_HOME/bin/startup.sh`
5. If you granted the user the `manager-gui` role, confirm that you can log into the manager webapp at `http://localhost:8080/manager/html` using the username and password of the manager account.

Configure Maven and Java Security Extension

Procedure

1. Update (or create if not present) your Maven repository settings.xml file for Maven deployment plugin authorization.

   Go to the .m2/settings.xml file of your operating system home directory (for Microsoft Windows, usually \Documents and Settings\<windows-user>; Linux would be /home/<user>) and add:

   ```
   <settings>
      <servers>
         <server>
            <id>myTomcat</id>
            <username>tomcat</username>
            <password>(defined in tomcat-users.xml configuration)</password>
         </server>
      </servers>
      ...
   </settings>
   ```

   Where "tomcat" above is the name of the user you granted the managerial role(s) to in the previous section.

2. Check if the Java Security Extension installed:

   To prevent the "Illegal key size or default parameters" exception, update your Java SDK by downloading the Java(TM) Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 6. Follow the README within that download for instructions on upgrading your JDK to support 256-bit encryption. (Another option is to reduce the encryption level of the sample to 128-bit by following the instructions in this sample's README file.) You can download the policy files from http://www.oracle.com/technetwork/java/javase/downloads/index.html. All versions of Java from 1.8.0_161 enable the unlimited strength policy files by default, see https://www.oracle.com/technetwork/java/javase/8u161-relnotes-4021379.html for more information.

Deployment and running

Procedure

1. From the root `jaxws-cxf-sts` folder, run `mvn clean install`. If no errors, can then run `mvn tomcat:deploy` (or `tomcat:undeploy` or `tomcat:redeploy` on subsequent runs as appropriate), either from the same folder (to deploy the STS and WSP at the same time) or separately, one at a time, from the `war` and `sts` folders.
2. Before proceeding to the next step, make sure you can view the following WSDLs: the CXF STS WSDL located at: `http://localhost:8080/DoubleItSTS/X509?wsdl` and the CXF WSP at `http://localhost:8080/doubleit/services/doubleitUT?wsdl`.
3. Navigate to the `client` folder and run `mvn clean install exec:exec`. You should see the results of three web service calls, with the client using X.509 authentication with the STS to get the SAML Assertion.