OSTwoUserManJavaDSIG

XML Digital Signatures

Signing a SAMLObject

Digitially signing a SAML object entails attaching and computing a Signature object to the object to be signed, where the Signature reflects the enveloped signature paradigm. Enveloped signatures are the only method formally prescribed in the XML Signature profile of the SAML specification. This is a 3-step process:

1. Create and attach a `Signature` object.
2. Marshall the object tree containing the `SAMLObject` which is to be signed.
3. Compute the the actual signature value contained within the `Signature` object.

Attaching a Signature to the SignableSAMLObject

The first step in signing a SAMLObject is to create a Signature object and attach it to the SAMLObject. Since all SAMLObjects that can be signed implement the interface `org.opensaml.common.SignableSAMLObject` the code to sign an object is always the same.

1. Create a `Signature` object using the `org.opensaml.xml.signature.impl.SignatureBuilder` (this is found in the XMLTooling library). This will likely be obtained from a builder factory. See Creating SAML Objects from Scratch.
2. Add the `Credential` containing the signing key using the `Signature#setSigningCredential(Credential)` method.
3. Add the signature method algorithm URI with the method `Signature#setSignatureAlgorithm(String)`. Note that the algorithm URI is dependent on the type of key contained with the signing credential.
4. Add the canonicalization method algorithm URI with the method `Signature#setCanonicalizationAlgorithm(String)`. Note that unless there is good reason to do otherwise, and the ramifications are understood, the recommended canonicalization method for SAML signature use cases is exclusive canonicalization (with or without comments).
5. If desired, add a `KeyInfo` containing information about the signature verification key using `Signature#setKeyInfo(KeyInfo)`. The `KeyInfo` may be created manually, or may be generated dynamically from the signing credential using a `KeyInfoGenerator`, usually obtained from a `KeyInfoGeneratorFactory` via a `KeyInfoGeneratorManager`.
6. Add the `Signature` to the SAMLObject using the `setSignature(Signature)` method. A `SAMLObjectContentReference` will automatically be added to the list of signature references exposed via `Signature#getContentReferences()`, so you should **NOT** explicitly add a reference to the signature. Note that the `Signature` contained within a signed element in SAML may contain only one `Reference` in the `SignedInfo`, per the SAML signature profile.

> **Dynamic Parameter Selection**
>
> The data inputs for steps 2-5 may be retrieved dynamically from an instance of `SecurityConfiguration` and populated on the `Signature` instance using the helper method `org.opensaml.xml.security.SecurityHelper#prepareSignatureParams`. For more usage info, see the Javadocs for that method.

Marshall the Object Tree

The signing operation operates on the underlying cached DOM representation of the object. Therefore, the SAML object to be signed must be marshalled before the actual signature computation is performed. For further information on marshalling, see Writing SAML Objects to XML .

Computing the Signature Value

After you have marshalled the XMLObject tree containing the SAMLObject which is to be signed, you'll have a SAML element with a digital signature child element populated with everything but the actual signature value. To signal that it is time to compute this, using the following steps:

1. Create an ordered `List` of all the `Signature` objects that need to be processed. This is necessary because one signature could contain the a reference to another and the second signature must be computed before the first signature or it will invalidate the first signature.
2. Pass the list to the `org.opensaml.xml.signature.Signer#signObject(List)` method.

> **Note**: if you only have one signature to be computed there is a convenience method `Signer#signObject(Signature)` to allow you to sign without creating a list.

Signing Examples

Here is an example of signing a SAML 2 `Assertion` with a `Credential` containing an RSA private key and using the RSA SHA1 algorithm, with explicit handling of `Signature` parameters:

```
Assertion assertion = createAssertion();
Credential signingCredential = getSigningCredential();

Signature signature = (Signature) Configuration.getBuilderFactory()
                        .getBuilder(Signature.DEFAULT_ELEMENT_NAME)
                        .buildObject(Signature.DEFAULT_ELEMENT_NAME);

signature.setSigningCredential(signingCredential);
signature.setSignatureAlgorithm(SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA1);
signature.setCanonicalizationAlgorithm(SignatureConstants.ALGO_ID_C14N_EXCL_OMIT_COMMENTS);
signature.setKeyInfo(getKeyInfo(signingCredential));

assertion.setSignature(signature);

try {
    Configuration.getMarshallerFactory().getMarshaller(assertion).marshall(assertion);
} catch (MarshallingException e) {
    e.printStackTrace();
}

try {
    Signer.signObject(signature);
} catch (SignatureException e) {
    e.printStackTrace();
}
```

Here is the same example, but with dynamic retrieval of parameters from the global `SecurityConfiguration`:

```
Assertion assertion = createAssertion();
Credential signingCredential = getSigningCredential();

Signature signature = (Signature) Configuration.getBuilderFactory()
                        .getBuilder(Signature.DEFAULT_ELEMENT_NAME)
                        .buildObject(Signature.DEFAULT_ELEMENT_NAME);

signature.setSigningCredential(signingCredential);

// This is also the default if a null SecurityConfiguration is specified
SecurityConfiguration secConfig = Configuration.getGlobalSecurityConfiguration();
// If null this would result in the default KeyInfoGenerator being used
String keyInfoGeneratorProfile = "XMLSignature";

try {
    SecurityHelper.prepareSignatureParams(signature, signingCredential, secConfig, keyInfoGeneratorProfile);
} catch (SecurityException e) {
    e.printStackTrace();
}

assertion.setSignature(signature);

try {
    Configuration.getMarshallerFactory().getMarshaller(assertion).marshall(assertion);
} catch (MarshallingException e) {
    e.printStackTrace();
}

try {
    Signer.signObject(signature);
} catch (SignatureException e) {
    e.printStackTrace();
}
```

Verifying the Signature on a Signed SAMLObject

Verifying a Signature with a Credential

Note: This is the lowest-level method of signature verification supported by the library, providing only cryptographic verification of the signature itself. For a higher-level API, see subsequent sections.

If your code had, through some mechanism such as a `CredentialResolver`, determined a `Credential` to use to attempt to verify the signature, you can perform the verification as follows:

1. Create an instance of a `org.opensaml.xml.signature.SignatureValidator`, passing the verification credential as the constructor argument.
2. Execute the `SignatureValidator#validate(Signature)` method, passing in the signature you wish to validate. If an exception is thrown the signature failed cryptographic verification and the exception will given the reason. If no exception is thrown then the cryptographic verification of the signature was successful.

> **Preventing Signature Denial of Service Attacks**
>
> In order to prevent certain types of denial-of-service attacks associated with signature verification, it is advisable to successfully validate the `Signature` with the `org.opensaml.security.SAMLSignatureProfileValidator` prior to attempting to cryptographically verify the signature.

> **Trust Establishment**
>
> Note that this signature verification mechanism does **NOT** establish trust of the verification credential, only that it successfully cryptographically verifies the signature. You **must** establish trust by other means. In particular, the `Credential` extracted from the signature's `KeyInfo` must **NOT** be trusted without additional evaluation.

Verifying a Signature with a SignatureTrustEngine

Trust engine implementations are available which combine cryptographic verification of the signature and trust establishment of the verification credential. These trust engines implement the `org.opensaml.xml.signature.SignatureTrustEngine` interface. Current implementations include:

- `org.opensaml.xml.signature.impl.ExplicitKeySignatureTrustEngine` - implements trust by retrieving trusted credentials from a `CredentialResolver` of trusted credentials.
- `org.opensaml.xml.signature.impl.PKIXSignatureTrustEngine` - implements trust by validating the verification credential (obtained from the signature's `KeyInfo`) using PKIX path validation rules. Trusted PKIX validation information is obtained from a `PKIXValidationInformationResolver`.
- `org.opensaml.xml.signature.impl.ChainingSignatureTrustEngine` - allows chaining of signature trust engines which implement different trust models or have different sources of trust material.

For more information on trust engines, see the user guide section Trust Engines.

Verifying a Signature with SAML 2 Metadata Information

SAML 2 metadata may contain keys, key names, and certificates which are associated with a particular role for a particular entity. Validating a signature against this information provides a significantly greater level of assurance in the signature as it proves not only that the signature is valid, but that the entity that generated it is interacting with the system in a role-appropriate manner.

Using SAML 2 metadata in conjunction with signature verification usually combines usage of a `SignatureTrustEngine` implementation with a trusted information resolver based on SAML 2 metadata, such as the `org.opensaml.security.MetadataCredentialResolver`. To perform validation based on this mechanism see the user guide section Trust Engines.

Signature Verification Examples

Here is a simple case of low-level cryptographically verifying the signature directly:

```
Response response = getResponse();

SAMLSignatureProfileValidator profileValidator = getSignatureProfileValidator();
try {
    profileValidator.validate(response.getSignature());
} catch (ValidationException e) {
    // Indicates signature did not conform to SAML Signature profile
    e.printStackTrace();
}

Credential verificationCredential = getVerificationCredential(response);
SignatureValidator sigValidator = new SignatureValidator(verificationCredential);
try {
    sigValidator.validate(response.getSignature());
} catch (ValidationException e) {
    // Indicates signature was not cryptographically valid, or possibly a processing error
    e.printStackTrace();
}
```

Here is a more complex example of verifying the signature with the explicit key signature trust engine, using trusted credentials obtained from SAML 2 metadata. The example assumes that the verifier is likely a SAML 2 SP, and is verifying the signature on a SAML 2 `Response` received from a SAML 2 IdP over the SAML 2.0 protocol:

```
//One-time init code here...
MetadataProvider mdProvider = getMetadataProvider();
MetadataCredentialResolver mdCredResolver = new MetadataCredentialResolver(mdProvider);
KeyInfoCredentialResolver keyInfoCredResolver =
    Configuration.getGlobalSecurityConfiguration().getDefaultKeyInfoCredentialResolver();
ExplicitKeySignatureTrustEngine trustEngine = new ExplicitKeySignatureTrustEngine(mdCredResolver,
keyInfoCredResolver);
storeSignatureTrustEngine(trustEngine);

// Individual message handling code here..
Response response = getResponse();

SAMLSignatureProfileValidator profileValidator = getSignatureProfileValidator();
try {
    profileValidator.validate(response.getSignature());
} catch (ValidationException e) {
    // Indicates signature did not conform to SAML Signature profile
    e.printStackTrace();
}

SignatureTrustEngine sigTrustEngine =  getSignatureTrustEngine();
CriteriaSet criteriaSet = new CriteriaSet();
criteriaSet.add( new EntityIDCriteria(response.getIssuer().getValue()) );
criteriaSet.add( new MetadataCriteria(IDPSSODescriptor.DEFAULT_ELEMENT_NAME, SAMLConstants.SAML20P_NS) );
criteriaSet.add( new UsageCriteria(UsageType.SIGNING) );

try {
    if (!sigTrustEngine.validate(response.getSignature(), criteriaSet)) {
        throw new Exception("Signature was either invalid or signing key could not be established as trusted");
    }
} catch (SecurityException e) {
    // Indicates processing error evaluating the signature
    e.printStackTrace();
}
```