OSTwoUserManJavaXMLEncryption

XML Encryption

Encrypting a SAMLObject

SAML 2 objects may be encrypted per the SAML 2 profile of the XML Encryption specification. Encryption consists of the following steps:

1. Specify data encryption parameters.
2. Specify key encryption parameters (if using one or more `EncryptedKey` elements to transport the data encryption key).
3. Create a SAML 2 `Encrypter` instance and set desired options.
4. Encrypt the desired SAML 2 `SAMLObject` instance or instances.

Some familiarity with the XML Encryption specification is assumed.

Specify Data Encryption Parameters

Data encryption parameters are specified by creating an instance of `org.opensaml.xml.encryption.EncryptionParameters`, and setting the appropriate properties.

- `algorithm` - specifies the symmetric block cipher used to encrypt the data. The value is an XML Encryption algorithm URI. This property may not be null. If not specified by the caller, an internal default will be used.
- `encryptionCredential` - specifies the symmetric encryption key that will be used to encrypt the data, in the form of a `org.opensaml.xml.security.credential.Credential`, containing a `javax.crypto.SecretKey`. It may be null, in which case a random data encryption key will be automatically generated and supplied via a minimal `Credential` instance.
- `keyInfoGenerator` - specifies an instance of `org.opensaml.xml.security.keyinfo.KeyInfoGenerator` which will be used to generate a `KeyInfo` element from the encryption `Credential`, which in turn will be included in the resultant `EncryptedData`. It may be null, in which case no `KeyInfo` will generated or included in the `EncryptedData`.

> **Dynamic Parameter Selection**
>
> Encryption parameter inputs may be retrieved dynamically from an instance of `SecurityConfiguration` and populated on an `EncryptionParameters` instance using the helper method `org.opensaml.xml.security.SecurityHelper#buildDataEncryptionParams`. For more usage info, see the Javadocs for that method.

Specify Key Encryption Parameters

Key encryption parameters are specified by creating an instance of `org.opensaml.xml.encryption.KeyEncryptionParameters`, and setting the appropriate properties.

- `algorithm` - specifies the key transport algorithm used to encrypt the data encryption key. The value is an XML Encryption algorithm URI. This property may not be null. There is no default, and it is the responsibility of the caller to ensure that the algorithm specified is consistent with the key encryption key specified in the `encryptionCredential` property.
- `encryptionCredential` - specifies the key encryption key that will be used to encrypt the data encryption key, in the form of a `org.opensaml.xml.security.credential.Credential`, containing either a `java.security.PublicKey` (for asymmetric key transport) or `javax.crypto.SecretKey` (for symmetric key wrap). This property may not be null, and it is the responsibility of the caller to ensure that the key encryption key specified is consistent with the algorithm specified in the `algorithm` property.
- `keyInfoGenerator` - specifies an instance of `org.opensaml.xml.security.keyinfo.KeyInfoGenerator` which will be used to generate a `KeyInfo` element from the key encryption `Credential`, which in turn will included in the resultant `EncryptedKey`. It may be null, in which case no `KeyInfo` will generated or included in the `EncryptedKey`.
- `recipient` - specifies the value of the `recipient` attribute that will be set on the resultant `EncryptedKey` element. It may be null, in which case no recipient attribute will be included.

`KeyEncryptionParameters` are specific to each intended recipient of the encrypted information. For the XML Encryption "multicast" use case, where multiple `EncryptedKey` elements are used to carry the data encryption to multiple recipients, multiple `KeyEncryptionParameters` should be created, each containing the appropriate parameters for each recipient.

> **Dynamic Parameter Selection**
>
> Key encryption parameter inputs may be retrieved dynamically from an instance of `SecurityConfiguration` and populated on a `KeyEncryptionParameters` instance using the helper method `org.opensaml.xml.security.SecurityHelper#buildKeyEncryptionParams`. For more usage info, see the Javadocs for that method.

Create a SAML 2 Encrypter

The main class used in SAML 2 encryption is an instance of `org.opensaml.saml2.encryption.Encrypter`. An instance is constructed by specifying via a constructor the `EncryptionParameters` and `KeyEncryptionParameters` to be used. Multiple constructor variants are available, depending on whether 0, 1, or 2+ `EncryptedKey` elements are to be generated.

Other options may then be set on the `Encrypter` instance to control how the encryption is performed. See the Javadocs for the `org.opensaml.saml2.encryption.Encrypter` and its superclass `org.opensaml.xml.encryption.Encrypter` for further details.

Encrypt the SAMLObject

The SAML 2 specialization of `Encrypter` supplies overloaded convenience methods for encrypting the types specified by the SAML 2 specification as capable of being encrypted: `Assertion`, `Attribute`, `NameID`, `BaseID`, and `NewID`. The return type of each method corresponds to the appropriate subtype of SAML 2 `EncryptedElementType` based on the original object that was encrypted.

Note that a SAML 2 `Assertion` may be encrypted as either an `EncryptedAssertion` or an `EncryptedID`, depending on the intended usage.

The generated `EncrytpedData` element will be a child of the `EncryptedElementType` subtype element. Any `EncryptedKey` elements will be placed as was specified in the `KeyEncryptionParameters`. In addition, forward and/or back references will be included between the `EncryptedData` and `EncryptedKey`, as specified in SAML 2 Errata item E43. See that document for further details.

Multiple SAML 2 objects may be encrypted with the same `Encrypter` instance, as long as the data and key encryption parameters supplied at construction time are the same for each encryption operation.

Encryption Examples

Here is an example of the encryption of a SAML 2 Assertion using the AES-128 symmetric block cipher. The encrypted data encryption key will be transported using the RSA-OAEP key transport algorithm, using the intended recipient's RSA public key. The single `EncryptedKey` will be placed as a peer of the `EncrytpedData`. The `EncryptedKey` will contain a `KeyInfo` containing information about the key encryption key that was used.

```
// The Assertion to be encrypted
Assertion assertion = getAssertion();

// Assume this contains a recipient's RSA public key
Credential keyEncryptionCredential = getKEKCredential();

EncryptionParameters encParams = new EncryptionParameters();
encParams.setAlgorithm(EncryptionConstants.ALGO_ID_BLOCKCIPHER_AES128);

KeyEncryptionParameters kekParams = new KeyEncryptionParameters();
kekParams.setEncryptionCredential(keyEncryptionCredential);
kekParams.setAlgorithm(EncryptionConstants.ALGO_ID_KEYTRANSPORT_RSAOAEP);
KeyInfoGeneratorFactory kigf =
    Configuration.getGlobalSecurityConfiguration()
    .getKeyInfoGeneratorManager().getDefaultManager()
    .getFactory(keyEncryptionCredential);
kekParams.setKeyInfoGenerator(kigf.newInstance());

Encrypter samlEncrypter = new Encrypter(encParams, kekParams);
samlEncrypter.setKeyPlacement(KeyPlacement.PEER);

try {
    EncryptedAssertion encryptedAssertion = samlEncrypter.encrypt(assertion);
} catch (EncryptionException e) {
    e.printStackTrace();
}
```

Decrypting an Encrypted SAMLObject

The steps involved in SAML 2 decryption are:

1. Obtain appropriate instances of `org.opensaml.xml.security.keyinfo.KeyInfoCredentialResolver`, used for resolving keys from `EncryptedData/KeyInfo` and/or `EncryptedKey/KeyInfo`.
2. Obtain an appropriate instance of `org.opensaml.xml.encryption.EncryptedKeyResolver`, used for resolving the correct `EncryptedKey` to be used in the context of decrypting a particular `EncryptedData` element.
3. Create a SAML 2 `Decrypter` instance and set desired options.
4. Decrypt the desired SAML 2 `SAMLObject` instance or instances.

Obtain KeyInfoCredentialResolver Instances

TODO

Obtain EncryptedKeyResolver Instance

TODO

Create a SAML 2 Decrypter

The main class used in SAML 2 encryption is an instance of `org.opensaml.saml2.encryption.Decrypter`. An instance is constructed by specifying via a constructor the `KeyInfoCredentialResolver` and `EncryptedKeyResolver` instances to be used. See the Javadocs for the superclass `org.opensaml.xml.encryption.Decrypter` for more details on the constructor arguments.

None of these 3 constructor arguments are mandatory in and of themselves. However, use cases will generally fall along at least one of 2 lines:

1. Recipient will decrypt the `EncryptedData` directly using a known, shared symmetric key. No `EncryptedKey` is present. In this case, the first argument `KeyInfoCredentialResolver` (`newResolver`) is necessary, but the second and third arguments are not used.
2. Recipient will decrypt a supplied `EncryptedKey`, which carries the encrypted data encryption key. Decryption of the `EncryptedKey` is accomplished by using either a private key corresponding to the public key used for encrypted key transport, or a shared symmetric key used for symmetric key wrap. In this case, it is typically necessary to supply the second argument `KeyInfoCredentialResolver` (`newKEKResolver`) and the `EncryptedKeyResolver` (`newEncKeyResolver`), but the first arg `KeyInfoCredentialResolver` (`newResolver`) is optional.

However, a specialized implementation of `KeyInfoCredentialResolver` which is designed to directly process `EncryptedKey` elements itself might supplant the need for a distinct KEK KeyInfo resolver and/or an encrypted key resolver.

Other options may then be set on the `Decrypter` instance to control how the encryption is performed. See the Javadocs for the `org.opensaml.saml2.encryption.Decrypter` and its superclass `org.opensaml.xml.encryption.Decrypter` for further details.

> If an object to be decrypted is signed with an enveloped signature (e.g. `Assertion`) and the signature is to be verified: You may need to call `Decrypter#setRootInNewDocument(true)` prior to decryption in order for signature verification to be successful on the decrypted `SignedSAMLObject`. For further details see the API Javadocs for `org.opensaml.xml.encryption.Decrypter`.

Decrypt the SAMLObject

The SAML 2 specialization of `Decrypter` supplies overloaded convenience methods for decrypting the types specified by the SAML 2 specification as capable of carrying encrypted SAML 2 elements: `EncryptedAssertion`, `EncryptedAttribute`, `EncryptedID`, and `NewEncryptedID`. The return type of each method corresponds to the appropriate SAML 2 element based on the original object that was encrypted.

> Note that a SAML 2 `EncryptedID` may carry either an encrypted `NameID`, `BaseID`, or `Assertion`. When decrypting an `EncryptedID`, it is up to the caller to determine the correct type of the decrypted `SAMLObject` that is returned, and cast it appropriately if desired.

Multiple SAML 2 objects may be decrypted with the same `Decrypter` instance, as long as the `KeyInfoCredentialResolver` and `EncryptedKeyResolver` instances supplied at construction time are appropriate for the multiple decryption operations. Alternatively, the `Decrypter` instance may be supplied with different `KeyInfoCredentialResolver` and `EncryptedKeyResolver` instances after construction by using the appropriate setter methods.

Decryption Examples

Here is a simple example of decryption where:

1. the data encryption key has been transported via an `EncryptedKey`, encrypted with the recipient's public key
2. the `PrivateKey` to use for decryption of the `EncryptedKey` is known in advance via some unspecified mechanism
3. the `EncryptedKey` is known in advance to have been carried within the `EncryptedData/KeyInfo`.

```
EncryptedAssertion encryptedAssertion = getEncryptedAssertion();

// This credential - obtained by some unspecified mechanism -
// contains the recipient's PrivateKey to be used for key decryption
Credential decryptionCredential = getDecryptionCredential();

StaticKeyInfoCredentialResolver skicr =
  new StaticKeyInfoCredentialResolver(decryptionCredential);

// The EncryptedKey is assumed to be contained within the
// EncryptedAssertion/EncryptedData/KeyInfo.
Decrypter samlDecrypter =
  new Decrypter(null, skicr, new InlineEncryptedKeyResolver());

try {
    Assertion assertion = samlDecrypter.decrypt(encryptedAssertion);
} catch (DecryptionException e) {
    e.printStackTrace();
}
```

Here is a more complex and realistic decryption case where:

1. the data encryption key has been transported via an `EncryptedKey`, encrypted with the recipient's public key
2. the `PrivateKey` to use for decryption of the `EncryptedKey` is **not** known in advance, and must be resolved from a store of local credentials, based on hints possibly provided in the `EncryptedKey/KeyInfo`
3. Several resolution mechanisms for finding the `EncryptedKey` must be supported simultaneously, including:
    a. inline within the `EncryptedData/KeyInfo`
    b. as a peer of the `EncryptedData` within the SAML 2 `EncryptedElementType`
    c. via a `RetrievalMethod` child of the `EncryptedData/KeyInfo`, which points via a same-document fragment reference to an `EncryptedKey` located elsewhere in the document.

```
//
// One-time init code here
//

// Collection of local credentials, where each contains
// a private key that corresponds to a public key that may
// have been used by other parties for encryption
List<Credential> localCredentials = getLocalCredentials();

CollectionCredentialResolver localCredResolver = new CollectionCredentialResolver(localCredentials);

// Support EncryptedKey/KeyInfo containing decryption key hints via
// KeyValue/RSAKeyValue and X509Data/X509Certificate
List<KeyInfoProvider> kiProviders = new ArrayList<KeyInfoProvider>();
kiProviders.add( new RSAKeyValueProvider() );
kiProviders.add( new InlineX509DataProvider() );

// Resolves local credentials by using information in the EncryptedKey/KeyInfo to query the supplied
// local credential resolver.
KeyInfoCredentialResolver kekResolver = new LocalKeyInfoCredentialResolver(kiProviders, localCredResolver);

// Supports resolution of EncryptedKeys by 3 common placement mechanisms
ChainingEncryptedKeyResolver encryptedKeyResolver = new ChainingEncryptedKeyResolver();
encryptedKeyResolver.getResolverChain().add( new InlineEncryptedKeyResolver() );
encryptedKeyResolver.getResolverChain().add( new EncryptedElementTypeEncryptedKeyResolver() );
encryptedKeyResolver.getResolverChain().add( new SimpleRetrievalMethodEncryptedKeyResolver() );

Decrypter samlDecrypter =
    new Decrypter(null, kekResolver, encryptedKeyResolver);

storeDecrypter(samlDecrypter);

// End init code

/* ........................... */

// Begin message processing code

Decrypter decrypter = getDecrypter();
EncryptedAssertion encryptedAssertion = getEncryptedAssertion();
try {
    Assertion assertion = decrypter.decrypt(encryptedAssertion);
} catch (DecryptionException e) {
    e.printStackTrace();
}
```