Providing Integrity and Confidentiality with the XML Security: Digital Signature and XML Encryption overview and usage examples

Draft version 0.1. - January 13, 2005

Yuri Demchenko <demch@science.uva.nl>

Abstracts

This document provides general overview of the XML Digital Signature and XML Encryption functionality and their use for providing document and message Integrity and Confidentiality for XML and Web Services based applications. Provided examples demonstrate using XML Signature and XML Encryption for and together with XACML Request/Response messages and SAML Assertions. Test programs are parts of the AAAuthreach package that provides simple API to Apache XML Security suite.

1 Introduction

XML Digital Signature and XML Encryption are two standard mechanisms used to provided integrity and confidentiality of XML documents and messages, in particular. Both of these mechanisms are integrated into SAML 2.0 and also used in SAML 1.1. Generally, XML Signature and XML Encryption can be added to any other XML format at the level of XML schema extension or to any valid XML document instance by using numerous available XML Security packages.

2 XML Digital Signature format and processing

This section provides general information about XML Signature format and some suggestions how it can be used together with SAML, XACML and XML Web Services in typical XML Web Services (WSA) and Open Collaborative Environment (OCE) applications.
2.1 XML Digital Signature format and processing

XML Signature has the following structure (see also picture below):

```xml
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI?>
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
    (<KeyInfo>)?
    (<Object ID?>)*
  </Signature>
```

There are three types of the XML Signature algorithms:

- **Enveloped XML Signature.**
  An enveloped signature is a signature of either an entire document or a document fragment, where the XML signature will itself be embedded within the signed document. An enveloped signature transform is always used to remove the signature structure from the signing process.

- **Enveloping XML Signature**
  An enveloping signature is a signature where the signed data is actually embedded within the XML signature structure: The XML signature specification provides the ability for arbitrary XML structures to be embedded within a signature, for this express purpose.

- **Detached XML Signature**
  A detached signature is a signature where the signed entities are separate from the actual signature fragment. The signed entities can be remote XML documents or remote non-XML documents. They can also be XML fragments located elsewhere in the same document as the XML signature.

It is important to mention that WS-I Basic Security Profile for Web Services recommends using detached signature and strongly discourages enveloping signature use, enveloped signature can be used but it provides limited functionality for signing the document and managing security components.
Creation of XML Signature includes the following steps:
1. Determine which resources are to be signed
2. Calculate the digest of each resource
3. Collect the Reference elements
4. Signing
5. Add key information
6. Enclose in a Signature element

XML Signatures verification includes the following steps:
- Verify the signature of the <SignedInfo> element
- Recalculate the digest of the <SignedInfo> element (using the digest algorithm specified in the <SignatureMethod> element)
• Use the public verification key to verify that the value of the `<SignatureValue>` element is correct for the digest of the `<SignedInfo>` element.

**If this step passes**

• Recalculate the digests of the references contained within the `<SignedInfo>` element and compare them to the digest values expressed in each `<Reference>` element's corresponding `<DigestValue>` element. XML Signature standard defines a number of transformations required for XML Signature signing and validation:
  - Canonicalisation
  - Base64
  - XPath Filtering
  - Envelope Signature Transform

XSLT Transformation

A canonicalisation is required to ensure the normalised presentation of the XML document in cases when XML format and schema validation allows options, like in using character encoding or attributes ordering. The *canonical form* of an XML document is physical representation of the document produced by the canonicalisation method that implies the following changes.

1) Encoding and characters
   • The document is encoded in UTF-8
   • Line breaks normalized to #xA on input, before parsing
   • Whitespace outside of the document element and within start and end tags is normalized
   • All whitespace in character content is retained (excluding characters removed during line feed normalization)

2) Elements and references
   • Character and parsed entity references are replaced
   • CDATA sections are replaced with their character content
   • The *XML declaration* and *document type declaration* (DTD) are removed
   • Empty elements are converted to start-end tag pairs

3) Attributes
   • Attribute values are normalized, as if by a *validating processor* Attribute value delimiters are set to quotation marks (double quotes)
   • Special characters in attribute values and character content are replaced by character references
   • Superfluous namespace declarations are removed from each element
   • Default attributes are added to each element

Lexicographic order is imposed on the namespace declarations and attributes of each element XML canonicalisation is defined in terms of the XPath definition of a node-set. If an XML document must be converted to a node-set, XPath REQUIRES that an XML processor be used to create the nodes of its data model to fully represent the document. The XML processor performs the following tasks in order:
   • normalize line feeds
   • normalize attribute values
   • replace CDATA sections with their character content
   • resolve character and parsed entity references
The input octet stream MUST contain a well-formed XML document, but the input need not be validated. The declarations in the document type declaration are used to help create the canonical form. This is a positive feature of the XML Signature transformation that signing program doesn’t need to validate a document.

2.2 XML Digital Signature examples

Examples below contain an Enveloped XML Digital Signature that signed the whole XML document URI = "" and the Subject element URI = "#subject" preserving in this way the document integrity as whole and the element that contains security token.

```xml
  <Subject Id="subject" SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject">
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject-id" DataTypes="http://www.w3.org/2001/XMLSchema#string" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>WHO740@users.collaboratory.nl</AttributeValue>
    </Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:token" DataTypes="http://www.w3.org/2001/XMLSchema#string" Id="subjecttoken" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>2SeDFGVHTY83ZXxEdsweOP8IokjyGHxVfHom90</AttributeValue>
    </Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:job-id" DataTypes="http://www.w3.org/2001/XMLSchema#string" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>JobID-XPS1-212</AttributeValue>
    </Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:role" DataTypes="http://www.w3.org/2001/XMLSchema#string" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>customer</AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id" DataTypes="http://www.w3.org/2001/XMLSchema#string" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>http://xps1.resources.collaboratory.nl/Phillips_XPS1</AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" DataTypes="http://www.w3.org/2001/XMLSchema#string" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>ControlExperiment</AttributeValue>
    </Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" DataTypes="http://www.w3.org/2001/XMLSchema#string" Issuer="admin@gaaa.collaboratory.nl">
      <AttributeValue>ViewExperiment</AttributeValue>
    </Attribute>
  </Action>
</AAARequest>
```

```xml
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:SignedInfo="http://www.w3.org/2000/09/xmldsig#"
  xmlns:CanonicalizationMethod="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"
  xmlns:SignatureMethod="http://www.w3.org/2000/09/xmldsig#rsa-sha1">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2000/09/xmldsig-c14n-20010315"/>
    <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <ds:Reference URI=""
    <ds:Transforms>
      <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
```
Listing 1. Example Enveloped XML Signature signed with the RSA private key and containing RSA public key.

Example below shows the `<ds:KeyInfo>` format for the DSA public key:

```xml
<ds:KeyInfo>
  <ds:X509Data>
    <ds:X509Certificate>
      MIICnTCCAkGA1UEBhMCTkxNDAYBgNVBAMTFEFBQXV0a59Mw8wDAYDVQQKExNhbG9yYXRpb24g
      +9wJGCGCSqGSIb3DQEBAQUAA4GAtCCAlGAoIBAgNVMEQwMGA0IBAQBh8yHIDCAKCIHv
      +9wJGCGCSqGSIb3DQEBAQUAA4GAtCCAf8wDQYJKoZIhvcNAQELBQADggIBATCCAlGAo
      +9wJGCGCSqGSIb3DQEBAQUAA4GAtCCAlGAoIBAQVGCBgQD34aCF1ps93su8q1w2uFe5e
      +9wJGCGCSqGSIb3DQEBAQUAA4GAtCCAf8wDQYJKoZIhvcNAQELBQADggIBAKHv
    </ds:X509Certificate>
  </ds:X509Data>
</ds:KeyInfo>
```

Example below shows the `<ds:KeyInfo>` format for the RSA public key:

```xml
<ds:KeyInfo>
  <ds:X509Data>
    <ds:X509Certificate>
      MIICdCBBgEKQYwDQJJKoZIhvcNAQEBgAwRBgNVMMYwZWQxFQgEBMFy58bCgYIKoZIhvcNAQEF
      a8gQgEBMFy58bCgYIKoZIhvcNAQEFa8gQgEBMFy58bCgYIKoZIhvcNAQEFa8gQgEBMFy58b
      CgYIKoZIhvcNAQEFa8gQgEBMFy58bCgYIKoZIhvcNAQEFa8gQgEBMFy58bCgYIKoZIhvcNA
      QEFa8gQgEBMFy58bCgYIKoZIhvcNAQEFa8gQgEBMFy58bCgYIKoZIhvcNAQEFa8gQgEBMFy58
      bCgYIKoZIhvcNAQEFa8gQgEBMFy58bCgYIKoZIhvcNAQEFa8gQgEBMFy58bCgYIKoZIhvcNAQ
    </ds:X509Certificate>
  </ds:X509Data>
</ds:KeyInfo>
```
Listing 2. Fragment `<ds:KeyInfo>` element with the DSA public key.

2.3 XML Signature Implementation suggestions for WSA/OCE:

For the purpose of WSA/OCE security assertions/tickets exchange and managing security context in the document-centric security model, the enveloped XML Signature provides necessary functionality:

- Enveloped signature is always placed under the root of document, however multiple Signatures are allowed in one document.
- Enveloped Signature can sign document in total and multiple elements defined by multiple Reference elements.
- To protect XML document integrity, it is recommended to sign the whole document with URI = "" and other critical security components such as security tokens, subject attributes and conditions that can identified by element's ID/Id, xpath or XPath expressions.
- Such voluminous component of the XML Signature element as KeyInfo can be reduced to KeyName if exchanging systems use pre-installed shared keys or public key certificates.
- The input document (in the form of octet stream or XML object) MUST be well-formed XML document, but the input needs not to be validated.
- XMLSig can be easily combined with the XML document validation what is highly recommended from the security point of view, if we are using the Enveloped XMLSig or providing a single container for the `<ds:Signature>` elements in the Detached XMLSig. The XMLSig elements or elements can ignored during the validation process, or added to the XML document schema.
- When considering use of one of available open source packages, it is important to take into account the following:
  
  (1) IBM's xss4j was popular at the early stage of XML Security dissemination but now it is announced not to be developed further;
Apache XML Security suite is widely used, including Sun’s XML Web Services Development Pack and Globus Toolkits 3.x.

Java XML Digital Signature API (JSR 105) potentially may become the most preferable package for processing XML Digital Signature as it implements Java community standard, it provides rich functionality and strict standards implementation, but its use currently limited by limited number of usage examples; another limitation is that it is not combined with the XML Encryption API that is being developed in the frame of JSR 106: XML Digital Encryption APIs but is not supported by any implementation.

3 XML Encryption

3.1 XML Encryption format and processing

XML Encryption datamodel:

```
<EncryptedData Id? Type? MimeTyp? Encoding?>
    <EncryptionMethod/>?
    <ds:KeyInfo>?
        <EncryptedKey>? # extension to XMLSig KeyInfo
        <AgreementMethod>?<ds:KeyName>?
        <ds:RetrievalMethod>?<ds:*>?
            #</ds:KeyInfo>?
    <CipherData> # envelopes or references the raw encrypted data
        <CipherReference URI?>?
            # points to the location of the raw encrypted data
    <EncryptionProperties>? # e.g., timestamp
</EncryptedData>
```

Figure 2 illustrates EncryptedData element structure. XML Encryption reuses ds:KeyInfo and ds:Transform elements from the XML Signature definition.
When encrypting data, for each EncryptedData and EncryptedKey the encryptor must:

1. Select the algorithm (and parameters)
2. Obtain and (optionally) represent the key
3. Encrypt the data
   - If the data is an "element" or element "content", obtain the octets by serialising the data in UTF-8; any other data must be serialised as octets
   - Encrypt the octets using the algorithm and key from steps 1 and 2
   - Provide type of presentation to indicate how to obtain and interpret the plaintext octets after decryption (e.g., MimeType="text/xml" or MimeType="image/png")
4. Build the EncryptedType (EncryptedData or EncryptedKey)
5. Process EncryptedData
   - If the Type of the encrypted data is "element" or element "content", then encryptor SHOULD be able to replace the unencrypted “element” or “content” with the EncryptedData element.
   - If the Type of the encrypted data is "element" or element "content", then encryptor MUST always be able to return the EncryptedData to the application.

Decryption processing rules include the following steps:

1. Locate encrypted data container xenc:EncryptedData that must have an attribute “type” that indicate may replace the element or
2. Process the element to determine the algorithm, parameters and ds:KeyInfo element to be used. If some information is omitted, the application MUST supply it.
3. Locate the data encryption key according to the ds:KeyInfo element, which may contain one or more children elements.
4. Decrypt the data contained in the CipherData element – depending on existence of CipherValue or CipherReference child elements
5. Process decrypted data of Type 'element' or element 'content'
   - The cleartext octet sequence (from step 3) is interpreted as UTF-8 encoded character data
   - The decryptor MUST be able to return the value of Type and the UTF-8 encoded XML character data. Validation on the serialized XML is NOT REQUIRED.
   - The decryptor SHOULD support the ability to replace the EncryptedData element with the decrypted 'element' or element 'content' represented by the UTF-8 encoded characters

6. Process decrypted data if Type is unspecified or is not 'element' or element 'content'.

XML Encryption implementation in Apache XML Security package require some additional JCE providers that are not normally shipped with the standard JSDK1.4. In this case, we need to use third party JCE (which is recommended from BouncyCastle at http://www.bouncycastle.org/download/jce-jdk13-114.jar). More about Java Cryptographic Architecture see Appendix E.

The XML Encryption specification lists a number of required and optional cryptographic algorithms including:
   - Block encryption – TripleDES, AES;
   - Stream encryption;
   - Key transport using public key encryption algorithms – RSA version 1.5, RSA-OAEP (Optimal Asymmetric Encryption Padding with RSA);
   - Key agreement used for derivation of a shared secret key based on a shared secret computed from certain types of compatible public keys from both the sender and the recipient – Diffie-Helman key value, Diffie-Helman key agreement,
   - Symmetric key wrap – Triple DES, AES;
   - Message digest used in key agreement method as a part of key derivation, with PSA-OAEP and with HMAC message authentication code – SHA1, SHA256, SHA512, RIPEMD-160;
   - Message authentication uses XML Signature method.

The Base64 encoding algorithm is used for binary encrypted data presentation. Every cipher must be Base64 encoded before it can be inserted into an XML document to be safely transferred over text based protocols.

3.2 XML Encryption examples

Examples below demonstrate encryption of the whole document (listing ) content or one of element determined by its tag name, in our example the Subject element of the generated test AAA:Request message. an Enveloped XML Digital Signature that signed the whole XML document URI = "" and the Subject element URI = "#subject" preserving in this way the document integrity as whole and the element that contains security token.

```xml
<AAA:AAARequest Id="CNLhashID" cnl:attr1="CNL2test" version="2.0" xmlns=""
xmlns:AAA="http://www.aaauthreach.org/ns/#AAA"
xmlns:cnl="http://www.telin.nl/ns/#cnl">
  <xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
    <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
      <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
```
Listing 3. Example XML Encryption of the content of the simplified AAA:Request message (using shared 3DES secret key for encryption of the data encryption key).

```xml
<AAA:AAARequest Id="CNLhashID#" cnl:attr1="CNL2test#" version="2.0" xmlns="" xmlns:AAA="http://www.aaauthreach.org/ns/#AAA" xmlns:cnl="http://www.telin.nl/ns/#cnl">
  <AAA:Subject Id="subject" xmlns="" xmlns:AAA="http://www.aaauthreach.org/ns/#AAA" xmlns:cnl="http://www.telin.nl/ns/#cnl">
    <xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <xenc:EncryptedKey xmlns="" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            <xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
          </xenc:CipherData>
        </xenc:EncryptedKey>
      </ds:KeyInfo>
      <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">qij9zJgKZp9RiJxYN1QJAN0vhjLJSMGVLD/dOQtmCsk</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </AAA:Subject>
</AAA:AAARequest>
```

Listing 4. Decrypted message that is equal to the original one.

```xml
<AAA:AAARequest Id="CNLhashID#" cnl:attr1="CNL2test#" version="2.0" xmlns="" xmlns:AAA="http://www.aaauthreach.org/ns/#AAA" xmlns:cnl="http://www.telin.nl/ns/#cnl">
  <AAA:Subject Id="subject" xmlns="" xmlns:AAA="http://www.aaauthreach.org/ns/#AAA" xmlns:cnl="http://www.telin.nl/ns/#cnl">
    <xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <xenc:EncryptedKey xmlns="" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            <xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">qij9zJgKZp9RiJxYN1QJAN0vhjLJSMGVLD/dOQtmCsk</xenc:CipherValue>
          </xenc:CipherData>
        </xenc:EncryptedKey>
      </ds:KeyInfo>
      <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">qij9zJgKZp9RiJxYN1QJAN0vhjLJSMGVLD/dOQtmCsk</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </AAA:Subject>
</AAA:AAARequest>
```
Listing 5. Example simplified AAA:Request message with the encrypted Subject element (the same encryption scenario).

```xml
  <xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          <xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">NxaCzuEaLizCB8nloGW3OVCKST34Wfa/ObP4MoJEUuA</xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedKey>
    </ds:KeyInfo>
    <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">FDH/hy8HAyYnp1p5Kzk/yzVQCUG3Qh8qT1JzXswogijtuWMcSN5cFuxARJ2g0PuRu8eBnзыPSuU1i4R5/jhNRJK3DSuhvurcTow14vcrRyXKquL7XKbpZLaTSg5rECML1lkqzM2/MXCl9u75g3jVlGF3I4x4znKXHWP4CF9g3Ty9qh71Ta/7bvaovOLWt/gKqVMB4qTm2jdRgshi183z3GTWwly/gn1XdmJYd/gm1u6990kywpmq3ExeA32red0dhbKRLmWlk2inmbyaVNPW4ePv1cv2mzYbv8P8nq9kgKRL2L1Gws7V26H6inMcunvJ/LYLPN4zeGF1i74S53e6XPNV5jHiqg/mc3CDcrPMeYgztZ85AhoGvKKS85+gscce/pXZ5m2fPdUSXXPGUIxereZ1pXyJ1cITZI7iTO7tf+z5/eCxxCV236An6EmAGkG7FZBNoRKhSPYbZbeWSy5aHeiZxx8TqGopcVE8WnpiahmX1cdctw/XsFy3fqngk/vgUZegw321101umBocLqMrM7707U7Hufuxvb5uxk6RUR0d1d0p6EW113MUY1md1gjVSVH2/0MyYrEmvEUvOQJ77zATVMYcX/BR7e99WYoWQCOqAg2jsQdlr/a080eKE/YE/EWIPM3fOR8HM6f7ett22geLgE1nEtMgNjKtjinMA5rd/C+GNCVCZmgqj1hhjFF6L6rjVBBw+vXq+Qf/xQFAS5LcpdzL/SXgk6OHur4u2UCVEXYXZ8F41KEVE201M4WG35YIOY4zTswbvTxdad458BN7h19uKhtsNNs6n60Vraro94dt5qR2FVAgUXAUsbFeYR1TrCqYoGqlStg80mqWB5p5y6b9pz4Xj2qRxveuK72K8FF1Ym9a9AtpdsxzXGfActAcnr2j045NwuLpaAfked01jgjcOscCTH4u6afPunLqJG3Rn11GjL991 Lambert/2T2MhUINUhralamewd+yKphzNS7h1mgUmbm01So+YdONX+Xe+xYr9GQnuJ3VcxspTWR981KX7sCSPNvG07Lwv/LpPPj7v+7xvej0n6EreHCTz2yU7IM4/4khvbdhz5x0/5/whTWnu9pCtbX8KAhFzdKwCjB0956BLLr22R97MhlfKFCNkC4/h/pdSKFDas/Is/PrintLEkRzwNwlqj212pCAl05Vix219AsRghT6+3CluF25gMFLZ021IrSMc1HwE6e+Bk+150Amw6BHAGBe8V1iToCkRydulysGZL335InXqyrW3g8mz0locpPNEXJAJ2azG4hVaiserHhx3y2ARAX9FY5ht4Kr4rmez/E/INGF بغ80+8h/F/sk0YJEEWq6E1EQVkbPopQyqbjfiz6Cdz3k5j55MdsdnChVWZ27E7Ece31WgNujnYjrgPtOpqmrPThmnVwa9jU+yMvZAIh6AYvKzhg9ortfXkB7567Tstg01n+v5e5In1JLCEX0hYQmRlariPN14myve5k+ruMVFyYYFJUuOA/k/duJ15dfvWGRgPyGJUXQJ1Uz1k12oSt3UtUJmn/THS2DndNa3IsAMVJ3esulQOqrHJh100d8sred/crOSbCjz2Wp+y19tCVCFF1IchOuKevEdWdJoMAd363yAruxwqgd7jlrj1Phbrj1xPzdodDbug3f4MzpPwfl13sGr6E5Oovv+GsSwrBwubh0/U/DEXrNhzehulidQQ2BrsrhrnH48BElBkolsKzXAtp/Em3eDj3ja4G9QY1H6rKWu48ZK9m6a5LbrepHP6lh9Prim2ryAUN5ryF9Qr90BfZnto/hdmsylzHGL71kCv2ZqL4T2y284f2XLfrKwDyysuraZU/Bmgx6wva6As+9diiuvAqXNuWyULZEU5VOFeUXQZk8nnsuy+yT+5SpBdpHc/saIj2cy6QipoaCPdB7Um0sGNg0qKbnLu/LrLrFNNbCOvDnfinf4965at6QgOEpK+AryR6P6e/QF/koxEpsyLyrUXBuA/Ao+10e3TVkXOnUXCN5s9H0q1V244LVjw7ycgV/9+19dxDMn5k4hE6sH82F7A2PzDkU2JF4P3Q003C0D0anw/aLOj2nxJ2H7JGF+61AcWm0bXlK5JvKIxKJmXmVUXa3J1Vgxs93ArtX0TmIzbylKVnPmDmdVWkrVijdobih8SmBdmW09YkxS91F9pux7037gIfYg093rBtU5QcKdr4BrYod/Akrjz9VCkgKwrzr9YS6wJeDa8Qbq0jGleKTJZTFieopCq9WMLJ```
Listing 6. Example real XACML AAA:Request message with the encrypted Subject element (the same encryption scenario).

```xml
<AAA:AAARequest Id="CNLhashID#" cnl:attr1="CNL2test#" version="2.0" xmlns=""
xmlns:AAA="http://www.aaauthreach.org/ns/#AAA"
xmlns:cnl="http://www.telin.nl/ns/#cnl">
<xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
<xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">0Zj2mUBrB2gzHgXr77TIEjz1yvIpjaczXkwUcarVehI7E5woun0k5d8ITF/JBYYmaJam4UMH4s5d
6/VPwGMwvRPX3HawXuJt1wMvKNyGy2T9DTWvDxSKG6wXT SSLt2csrMLiTWCoOvIQSE1UP/9cyYr
WYSW5ukX6j4Bpqdxyg</xenc:CipherValue>
</xenc:CipherData>
</xenc:EncryptedKey>
</ds:KeyInfo>
</xenc:EncryptedData>
</AAA:AAARequest>
```

Listing 7. Example XML Encryption of the content of the simplified AAA:Request message (using shared RSA public key for encryption of the data encryption key).

```xml
<AAA:AAARequest Id="CNLhashID#" cnl:attr1="CNL2test#" version="2.0" xmlns=""
xmlns:AAA="http://www.aaauthreach.org/ns/#AAA"
xmlns:cnl="http://www.telin.nl/ns/#cnl">
<xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
<xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:CipherValue xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">0Zj2mUBrB2gzHgXr77TIEjz1yvIpjaczXkwUcarVehI7E5woun0k5d8ITF/JBYYmaJam4UMH4s5d
6/VPwGMwvRPX3HawXuJt1wMvKNyGy2T9DTWvDxSKG6wXT SSLt2csrMLiTWCoOvIQSE1UP/9cyYr
WYSW5ukX6j4Bpqdxyg</xenc:CipherValue>
</xenc:CipherData>
</xenc:EncryptedKey>
</ds:KeyInfo>
</xenc:EncryptedData>
</AAA:AAARequest>
```

3.3 XML Encryption implementation suggestions for WSA/OCE:

XML Encryption can be used to encrypt sensitive parts of CNL Job description that provides bot BA and TA for CNL/OCE secure environment. It can also provide a method to exchange shared secrets like in case of establishing trusted relations between participating in CNL parties or VO members. Actually, XML Encryption is a part of WS-SecureConversation mechanisms.
• XML Encryption uses a security paradigm slightly different from XML Signature. XML Signature uses public key or symmetric crypto algorithm for signing quite straightforward. Major suggested/recommended XML Encryption procedure uses symmetric data encryption key (dek), often generated instantly, for encrypting data and next supplies this (instantly generated) data encryption key in the encrypted form using another symmetric key (kek) or public key of the recipient. The reason for this is that the symmetric key and data encrypted with the symmetric key have smaller size.

• Recommended in XML Encryption specification and by WS-I are the following algorithms: AES as a data encryption algorithm and key; TripleDES key wrap or RSA version 1.5 key transport for dek encryption.

• To encrypt XML document or element with Apache XML Security package for Java, it is required to provide the context document and an element to be encrypted. It important that the DOM methods used to get the element is namespace aware as both XML Encryption and XML Signature are namespace aware. This in particular related to the methods 
getElementsByTagNameNS(URI-NS, ElementName) which is namespace aware and 
getElementById(IdString) which is not namespace aware. However, the element can be identified by Id by applying ds:Reference URI transformation.

• When considering adding XML Encryption to an application, the XML document validation issues should be considered. Due to the fact that XMLEnc replaces the element or element's content, the validation against original document schema may fail. To avoid this situation, the validating parser should be XML Encryption aware, what in its own turn should carefully analysed from the security point of view, or the option of the encryption element of content should added to the XML document schema, as it is done in the SAML 2.0.

• Recommendation regarding existing XML Encryption Java packages are almost the same as for XML Digital Signature (see section 2.3 above), except that actually choice can be limited to IBM’s xss4j either Apache XML Security suite, where we can recommend Apache’s one.

4 Java Cryptographic Architecture (JCA)

The Java Cryptographic Architecture (JCA) provides all necessary basic cryptographic services required for PKI components management and XML security.

Java offers complete support for cryptography. For this purpose, there are several packages inside J2SE, covering all the main features of security architecture such as access controls, signatures, certificates, key pairs, key stores, and message digests.

The primary principle of JCA design is to separate cryptographic concepts from algorithmic implementations, so that different vendors can offer interoperable tools within the JCA framework.

Implementation independence is achieved using a "provider"-based architecture. The term Cryptographic Service Provider (used interchangeably with "provider" in this document) refers to a package or set of packages that implement one or more cryptographic services, such as digital
signature algorithms, message digest algorithms, and key conversion services. JCA defines a series of Engine classes, where each Engine provides a cryptographic function.

**JCA Engine classes**

An *engine class* defines a cryptographic service in an abstract fashion (without a concrete implementation). Engine classes may have different implementation, but at the Engine API level they are all the same.

The following engine classes are defined in Java 2 SDK:

- **MessageDigest**: used to calculate the message digest (hash) of specified data.
- **Signature**: used to sign data and verify digital signatures.
- **KeyPairGenerator**: used to generate a pair of public and private keys suitable for a specified algorithm.
- **KeyFactory**: used to convert opaque cryptographic keys of type Key into key specifications (transparent representations of the underlying key material), and vice versa.

**Note.**
- **CertificateFactory**: used to create public key certificates and Certificate Revocation Lists (CRLs).
- **KeyStore**: used to create and manage a keystore. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.
- **AlgorithmParameters**: used to manage the parameters for a particular algorithm, including parameter encoding and decoding.
- **AlgorithmParameterGenerator**: used to generate a set of parameters suitable for a specified algorithm.
- **SecureRandom**: used to generate random or pseudo-random numbers.
- **CertPathBuilder**: used to build certificate chains (also known as certification paths).
- **CertPathValidator**: used to validate certificate chains.
- **CertStore**: used to retrieve Certificates and CRLs from a repository.

**Java Cryptographic Extensions (JCE)**

All independent (third party) vendor implementations of cryptographic algorithms are called Java Cryptographic Extensions (JCE). Sun Microsystems has also provided an implementation of JCE. Whenever we use JCE, we need to configure it with JCA. For this, we need to do the following:

1. Add the address of the jar file to configure the provider (all JCE implementations are called providers) in the CLASSPATH environment variables.

2. Configure the provider in the list of your approved providers by editing the `java.security` file. This file is located in JavaHome/jre/lib/security folder. The following is the syntax to specify the priority: `security.provider.<n>=<masterClassName>`. Here, `n` is the priority number (1, 2, 3, etc.). `MasterClassName` is the name of master class to which the engine classes will call for a specific algorithm implementation. The provider's documentation will specify its master class name. For example, consider the following entries in a java.security file:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
```
These entries mean that the engine class will search for any algorithm implementation in the above mentioned order. It will execute the implementation found first. After these simple steps, we are all set to use JCA/JCE in our XML Encryption application.

Key management

A database called a "keystore" can be used to manage a repository of keys and certificates. A certificate is a digitally signed statement from one entity, saying that the public key of some other entity has a particular value. A keystore is available to applications that need it for authentication or signing purposes.

Applications can access a keystore via an implementation of the KeyStore class, which is in the java.security package. A default KeyStore implementation is provided by Sun Microsystems. It implements the keystore as a file, using a proprietary keystore type (format) named "JKS".

Applications can choose different types of keystore implementations from different providers, using the getInstance factory method supplied in the KeyStore class.

Currently, there are two command-line tools that make use of KeyStore: keytool and jarsigner, and also a GUI-based tool named policytool. Below are examples how to generate and save RSA key pair in keystore (changing –keyalg parameter to DSA will generate DSA keys):

1) generate RSA key pair with alias “cnl01”

```
keytool -genkey -alias cnl01 -keyalg RSA -dname "CN=AAAuthreach Security,
O=Collaboratory.nl, C=NL" -keypass xmlsecurity -keystore xmlsec/keystore1xmlsec.jks
-storepass xmlsecurity -storetype %STORETYPE%
```

2) two step procedure of extracting interesting key from the existing keystore1*, generating X.509 certificate and storing it in a new keystore2*

```
echo Copying certificate of RSA public key...
keytool -export -alias cnl01 -file xmlsec/cnl01.cer -keystore
xmlsec/keystore1xmlsec.jks -storepass xmlsecurity -storetype %STORETYPE%
keytool -import -noprompt -alias cnl01 -file xmlsec/cnl01.cer -keystore
xmlsec/keystore2xmlsec.jks -storepass xmlsecurity -storetype %STORETYPE%
```

All these functions are also available programmatically via KeyStore class that supplies well-defined interfaces to access and modify the information in a keystore.

5 References


