

nehta

Example Technical Implementation of XML Secured Payload Profile

Java

Version 1.0 — 1 December 2008

Release

National E-Health Transition Authority Ltd

Level 25

56 Pitt Street

Sydney, NSW, 2000

Australia.

www.nehta.gov.au

Disclaimer

NEHTA makes the information and other material (“Information”) in this document available in good faith but without any representation or warranty as to its accuracy or completeness. NEHTA cannot accept any responsibility for the consequences of any use of the Information. As the Information is of a general nature only, it is up to any person using or relying on the Information to ensure that it is accurate, complete and suitable for the circumstances of its use.

Document Control

This document is maintained in electronic form. The current revision of this document is located on the NEHTA Web site and is uncontrolled in printed form. It is the responsibility of the user to verify that this copy is of the latest revision.

Copyright © 2008, NEHTA.

This document contains information which is protected by copyright. All Rights Reserved. No part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without the permission of NEHTA. All copies of this document must include the copyright and other information contained on this page.

Table of contents

Table of contents	iii
Document information	v
Change history	v
1 Introduction	1
1.1 Background	1
1.2 Purpose	1
1.3 Scope	1
1.4 Intended Audience	1
1.5 Definitions, acronyms, abbreviations	2
1.5.1 Acronyms	2
1.5.2 XML namespaces	2
1.6 Style Conventions	2
1.7 Document overview	3
2 Signed Payload	4
2.1 Conceptual overview	4
2.1.1 Digital signature	4
2.1.2 XML Signature	4
2.1.3 Signed payload container	5
2.2 Libraries used	6
2.3 Signing Process	7
2.3.1 Create the container document	7
2.3.2 Import the payload	8
2.3.3 Add the ID attribute	9
2.3.4 Normalize the document	9
2.3.5 Get an XMLSignatureFactory instance	9
2.3.6 Create the Reference object	9
2.3.7 Create the SignedInfo object	10
2.3.8 Create the Signatures	11
2.4 Verification Process	12
2.4.1 Get the signature elements	12
2.4.2 Get an XMLSignatureFactory instance	13
2.4.3 Check the signatures	13
2.4.4 Extract the payload	14
3 Encrypted Payload	15
3.1 Conceptual overview	15
3.1.1 Encryption	15
3.1.2 XML Encryption	15
3.1.3 Encrypted payload container	16
3.2 Libraries used	17
3.3 Encryption process	18
3.3.1 Create the container document	18
3.3.2 Generate the session key	19
3.3.3 Generate the reference ID	20
3.3.4 Encrypt the data	20
3.3.5 Encrypt session key for each receiver	21
3.4 Decryption process	22
3.4.1 Find the encrypted key	22
3.4.2 Find encrypted data	24
3.4.3 Determine data encryption algorithm	24
3.4.4 Decrypt session key	24
3.4.5 Decrypt payload	24
4 Composite Payload	25

4.1	Conceptual Overview.....	25
4.2	Signed Encrypted Payload.....	25
5	Secured payloads with Web services.....	26
5.1	Secured payload as XML Schema "any".....	26
5.2	Secured payload as XML Schema "element".....	27
	Appendix A: References.....	29
	Appendix B: Key management.....	30
B.1	Key store types.....	30
B.2	Tools.....	30
	Appendix C: Installation.....	31
C.1	Java Development Kit.....	31
C.2	Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files	31
C.3	Apache XML Security API.....	31
	Appendix D: Helper classes.....	32
D.1	CertificateKeyPair.....	32
D.2	XmlUtils.....	32

Document information

Change history

Version	Date	Comments
1.0	2008-12-01	Release

This page is intentionally left blank.

1 Introduction

1.1 Background

The National E-Health Transition Authority (NEHTA) has recommended Web services as the mechanism for application-to-application communications in Australia's e-health environment. Web services use the Extensible Markup Language (XML) as the format for representing data.

It has been recognized that some communication patterns may involve using an intermediate party to store data for the intended receiver. To guarantee the privacy and integrity of sensitive data, the *XML Secured Payload Profile* [XSP2008] has been defined to allow the digital signing and encryption of data.

1.2 Purpose

This document provides explanations of the signing and encrypting processes, and implementation examples of how a signed and encrypted payload is prepared. The implementation examples use specific toolkits: *Sun JDK 6* [JDK6] for signing, and the *Apache XML Security* [AXMLSEC] library for encryption.

The main purpose of the document is to support the understanding and interpretation of the criteria in the *XML Secured Payload Profile* [XSP2008]. However, it can also assist programmers who are learning how to use the tools.

This document is provided for educational purposes only. The method it describes is only one approach; there might be other valid approaches. The code samples in this document are designed for simplicity and ease of understanding, rather than robustness and reuse. They are not written for use in a production system.

1.3 Scope

This document only covers implementation in Java. Also available is an example technical implementation document covering the Windows .NET Framework. That other example technical implementation can interoperate with this implementation, but it will not be discussed in this document. These examples are not an endorsement of these platforms by NEHTA.

1.4 Intended Audience

This document is intended for:

- Software developers.

It is expected that the reader is familiar with programming using Java, and has an understanding of XML, XML Signature, XML Encryption and Public Key Infrastructure (PKI) security using X.509 certificates.

The reader is also expected to be familiar with the *XML Secured Payload Profile* [XSP2008]. The criteria from [XSP2008] are referred to by their criterion number (e.g. "XS 3.1.1.1-1").

1.5 Definitions, acronyms, abbreviations

1.5.1 Acronyms

API	Application Programming Interface
CA	Certificate Authority
DOM	Document Object Model
JAXB	Java API for XML Binding
JAX-WS	Java API for XML Web Services
JDK	Java Development Kit
JRE	Java Runtime Environment
JSE	Java Standard Edition
PKI	Public Key Infrastructure

1.5.2 XML namespaces

This document refers to XML elements belonging to different namespaces. For ease of understanding, this document uses the same prefix for elements belonging to the same namespace.

The table below shows the prefixes and namespaces used by this document. Note that the prefixes were chosen to follow convention. There is no technical requirement to use a particular prefix for a namespace.

Prefix	Namespace
sp	http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0 (Signed payload [XSP2008])
ep	http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0 (Encrypted payload [XSP2008])
ds	http://www.w3.org/2000/09/xmlsig# (XML Signature [XSXSD2002])
xenc	http://www.w3.org/2001/04/xmlenc# (XML Encryption [XEXSD2002])

1.6 Style Conventions

This document uses the following style conventions:

<i>Italics</i>	<ul style="list-style-type: none"> • Document titles • Program names, tool names • File names, directory paths • URLs
Monospace	<ul style="list-style-type: none"> • XML fragments, names of elements and types,

	<p>namespaces</p> <ul style="list-style-type: none">• Code fragments, names of classes, methods, and fields• Assemblies, packages• Command-line calls and arguments• Configuration properties
Monospace + Bold	<ul style="list-style-type: none">• Emphasis for within XML and code fragments

1.7 Document overview

Chapter 2 describes how to create and verify signed payloads.

Chapter 3 describes how to create and decrypt encrypted payloads.

Chapter 4 describes how to create composite payloads.

Chapter 5 described how to use secured payloads within Web services.

Appendix A: lists references.

Appendix B: describes tools for key management.

Appendix C: provides instructions for software installation.

Appendix D: provides code listings for helper classes.

2 Signed Payload

The *XML Secured Payload Profile* [XSP2008] provides a mechanism called a signed payload that uses *XML Signature* [XDSIG2002] to ensure integrity of XML data and to authenticate the creators of the data.

This chapter explains how to create and check signed payloads that are conformant to the *XML Secured Payload Profile* [XSP2008].

Section 2.1 provides an introduction to the technologies and specifications relevant to signed payloads. It is a conceptual overview that is independent of implementations.

The Java JDK 6 has an API that implement *XML Signature*, which can be used to create and process signed payloads. Section 2.2 introduces this API and others libraries used by the example code. Section 2.3 provides example code to create a signed payload. Section 2.4 provides example code to verify a signed payload and extract the original XML data.

2.1 Conceptual overview

2.1.1 Digital signature

Signing is the process of creating a digital signature on plaintext to authenticate the signer and to ensure the integrity of the plaintext.

The process of signing is asymmetric. A private key is used to create the signature and a corresponding public key is used to verify the signature.

To create a digital signature, the signer first calculates a digest value on the data being signed. A digest value is used instead of the actual data for efficiency. A signature value is calculated using a signature algorithm with the digest value and the signer's private key.

To validate a digital signature, the receiver uses the signer's public key to retrieve the digest value from the signature value. The receiver also separately calculates the digest value on the data. The digest values from the signature value and from the receiver's calculation must match for the signature to be valid.

2.1.2 XML Signature

Digital signatures can be created for different types of data, not necessarily XML data. The *XML Signature* specification [XDSIG2002] defines how digital signatures are applied to XML data. It specifies the processes for creating and verifying digital signatures on XML data and a way to represent digital signatures in an XML format.

An XML Signature is created by the following steps:

1. A digest value is calculated for each XML data fragment being signed. This involves first applying a set of transforms to the XML fragment, then calculating the digest on the transformed XML fragment. The transformations ensure the XML fragment is in a normalized form. This usually includes XML canonicalization. The information from this step is represented using a `ds:Reference` element.
2. The `ds:Reference` elements from the previous stage are added to a `ds:SignedInfo` element. A digest value is calculated on the `ds:SignedInfo` element which involves first applying XML canonicalization. This calculated digest value is signed using the signer's private key to create the `ds:SignatureValue` element. A `ds:KeyInfo` element is used to specify which key was used to create the signature.

The `ds:SignedInfo`, `ds:SignatureValue` and `ds:KeyInfo` elements are added to a `ds:Signature` element which is the resulting signature.

An XML Signature is validated by the following steps:

3. A digest value is calculated for each `ds:Reference` element within the signature. This involves applying the transforms specified in the reference, then calculating the digest value on the transformed XML fragment. The calculated digest value is compared to the one that is within the `ds:Reference` element. When they don't match, the signature validation fails.
4. A digest value is calculated on the `ds:SignedInfo` element. This involves first applying canonicalization on this element. The digest value of the `ds:SignedInfo` element is retrieved from the signature value using the signer's public key. This digest value is compared with the calculated digest value. When they don't match, the signature validation fails.

2.1.3 Signed payload container

The use of *XML Signature* [XDSIG2002] doesn't guarantee that one implementation can validate the signed XML data of another implementation since there are several options when creating an XML Signature. Therefore, the *XML Secured Payload Profile* was created [XSP2008].

The *XML Secured Payload Profile* specification defines a way of using XML Signature in order to enable interoperability when exchanging signed XML data. For instance, it specifies what algorithms to use, how data is referenced and how keys are referenced. It also defines how the XML signatures and data are to be placed in a container XML document.

The example listing below shows a signed payload container conforming to the *XML Secured Payload Profile* specification [XSP2008].

```
<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
  <sp:signatures>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        <ds:SignatureMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
        <ds:Reference URI="#a0ede6c4-21e4-44d5-93b3-4b2f02f9e1f8">
          <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:Transforms>
          <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
          <ds:DigestValue>Orn5miXZzb9EYhwcMg4ZTWavRds=</ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>
ffzZtvV4ms46VbZRLM6r0eMoOirKpwd1kOAVxhLqs/TPgtIjDqgX4HMqmRbjNibAtO4KvMnLkx6NUq3PbN
FY0DZCf9xTEw2HTOCSRvRpxlL0DPbVc5Jy1U0v3e1hbcglCmhqfRyI9003b3se9WpF/nkDSZGAov2NmhWn
bnNcqjE=
      </ds:SignatureValue>
      <ds:KeyInfo>
        <ds:X509Data>
<ds:X509Certificate>MIICBjCCAW+gAwIBAgIBBjANBgkqhkiG9w0BAQUFADAeMRwwGgYDVQQDExNORU
hUQSBFSULXUyBkZW1vIENBMB4XDTA3MDYyMjAwMDI1NFoXDTE3MDYyOTAwMDI1NFowFzEVMBMGA1UEAxQM
amF4d3NfY2xpZW50MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDVNtVihAhN76KhBevo/YTzBx1oB7
K8YqRhfG3ca/Y9qFv1Mk6tUNjC9LxBqtLcQbcLpeZOFsKumtygvvZUBAZlpTR7qMOeHcFMYCp1sVS4mTo
TRt8P33au2x6VvTj7AWaxr8Di1jv13hB/luPKRuf1yw0hV7mzvSkc1wHVBBn0QIDAQABo1swWTAMBgNVHR
MBAf8EAjAAMB0GA1UdDgQWBBS1A5kUs7rHCR8wZEX7Nb4m3k1zOzAfBgNVHSMEGDAWgBQL0DcCBAjrOSF5
Usiwo0701htHBzAJBgNVHREEEAjAAMA0GCSqGSIb3DQEBAQUAA4GBAGEP1kuOg5RftVWrfP+PSgHueUugWH
hOqUsUB5w3OPYhkIawVfbrXu4cQ+wSo96yiP/89xsxTwwiWa0LQ02xmEZWf2F9Frc02Ni9nZ1lulREtrd
5Huino80GmEB4AJWdhGV0GAT45Ze/tuVg+Xa+YmvHiuYseLVGeirnEOmoPS2</ds:X509Certificate>
        </ds:X509Data>
      </ds:KeyInfo>
    </ds:Signature>
  </sp:signatures>
  <sp:signedPayloadData id="a0ede6c4-21e4-44d5-93b3-4b2f02f9e1f8">
    <pathologyReport xmlns="http://www.example.org">
```

```

<creator id="1234"/>
<receiver id="4321"/>
<patient id="5678"/>
<results>
  <test name="blood pressure" units="mmHg">90-119</test>
</results>
</pathologyReport>
</sp:signedPayloadData>
</sp:signedPayload>

```

2.2 Libraries used

JSR-105 is a standard Java API that implements the *XML Signatures* specification [XDSIG2002]. JDK 6 contains an implementation of the JSR-105 API. This document describes how to create and validate XML signatures using this API. The packages used from this XML Signature API are:

- `javax.xml.crypto`,
- `javax.xml.crypto.dsig`,
- `javax.xml.crypto.dsig.keyinfo` and
- `javax.xml.crypto.spec`.

Classes from the general security libraries of the JDK are used to load keys and perform trust chain verification. The general security packages used are:

- `java.security` and
- `java.security.cert`.

XML data is represented and processed using the Document Object Model (DOM). The packages used for loading and processing DOM are:

- `javax.xml.crypto.dsig.dom`,
- `javax.xml.parsers`, and
- `org.w3c.dom`.

The following import statements apply to the code listings in sections 2.3 and 2.4:

```

import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;

import java.security.cert.CertPathBuilder;
import java.security.cert.PKIXBuilderParameters;
import java.security.cert.X509Certificate;
import java.security.cert.X509CertSelector;

import javax.xml.crypto.dsig.CanonicalizationMethod;
import javax.xml.crypto.dsig.DigestMethod;
import javax.xml.crypto.dsig.Reference;
import javax.xml.crypto.dsig.SignatureMethod;
import javax.xml.crypto.dsig.SignedInfo;
import javax.xml.crypto.dsig.XMLSignature;
import javax.xml.crypto.dsig.XMLSignatureException;
import javax.xml.crypto.dsig.XMLSignatureFactory;

import javax.xml.crypto.dsig.dom.DOMSignContext;
import javax.xml.crypto.dsig.dom.DOMValidateContext;

import javax.xml.crypto.dsig.keyinfo.KeyInfo;
import javax.xml.crypto.dsig.keyinfo.KeyInfoFactory;
import javax.xml.crypto.dsig.keyinfo.X509Data;

import javax.xml.crypto.dsig.spec.C14NMethodParameterSpec;
import javax.xml.crypto.dsig.spec.TransformParameterSpec;

import javax.xml.crypto.MarshalException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

```

```
import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

2.3 Signing Process

The steps involved in creating a signed payload container that complies with the *XML Secured Payload Profile* [XSP2008] are:

- Creating the container document;
- Importing the payload into the container;
- Adding the ID attribute onto the payload;
- Normalizing the document;
- Getting an XMLSignatureFactory instance;
- Creating the Reference object;
- Creating the SignedInfo object; and
- Creating the signatures over the identified payload.

The inputs to creating the signed payload container are:

Document payloadDoc

A `org.w3c.dom.Document` containing the payload to be signed. This can be any XML document.

List<CertificateKeyPair> keyPairs

A list of one or more X.509 certificates with their corresponding private key. The private keys are used to sign the payload. `CertificateKeyPair` is a helper class that is used to store a pair of `java.security.cert.X509Certificate` and `java.security.PrivateKey`. This class is provided in appendix D.1.

The output of creating the signed payload container is:

A `org.w3c.dom.Document` containing XML structured according to the signed payload XML Schema defined in *XML Secured Payload Profile* [XSP2008].

Note that any error checking in the example code in this section has been omitted.

2.3.1 Create the container document

The *XML Secured Payload Profile* [XSP2008] specifies in criterion XS 3.1.1.1-1 an XML structure that acts as a container for the signed data and one or more digital signatures. The section explains how to create this XML structure in a DOM Document object.

2.3.1.1 Create an empty document

An empty DOM document is created by calling the `newDocument` method on an instance of the `javax.xml.parsers.DocumentBuilder` class.

For XML encryption to work correctly, the DOM document that is created must support XML namespaces. This is ensured by setting `namespaceAware` to `true` on the factory that creates the `DocumentBuilder` instance.

```

DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
docBuilderFactory.setNamespaceAware(true);
DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();

Document containerDoc = docBuilder.newDocument();

```

2.3.1.2 Create the sp:signedPayload element

The sp:signedPayload element is the root element of the container. This element is created and added to the container document.

```

Element signedPayloadElem = containerDoc.createElementNS(
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0",
    "sp:signedPayload");
containerDoc.appendChild(signedPayloadElem);

```

This results in the container document containing the following structure:

```

<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
</sp:signedPayload>

```

2.3.1.3 Create the sp:signatures element

The sp:signatures element is used to hold the signature that will be created on the payload. The sp:signatures element is created and added to the sp:signedPayload root element.

```

Element signaturesElem = containerDoc.createElementNS(
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0",
    "sp:signatures");
signedPayloadElem.appendChild(signaturesElem);

```

This results in the container document containing the following structure:

```

<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
  <sp:signatures />
</sp:signedPayload>

```

2.3.1.4 Create the sp:signedPayloadData element

The sp:signedPayloadData element is used to hold the payload data being signed. The sp:signedPayloadData element is also created and added to the sp:signedPayload root element.

```

Element signedPayloadDataElem = containerDoc.createElementNS(
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0",
    "sp:signedPayloadData");
signedPayloadElem.appendChild(signedPayloadDataElem);

```

This results in the container document containing the following structure:

```

<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
  <sp:signatures />
  <sp:signedPayloadData />
</sp:signedPayload>

```

2.3.2 Import the payload

The payload, passed in as an argument in payloadDoc, must be imported and added as a child element of the sp:signedPayloadData element.

```

Node payloadNode = containerDoc.importNode(payloadDoc.getDocumentElement(), true);
signedPayloadDataElem.appendChild(rawPayloadNode);

```

After this step, the container document has the following XML:

```

<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
  <sp:signatures />
  <sp:signedPayloadData>
    <pathologyReport xmlns="http://www.example.org">
      <creator id="1234"/>
      <receiver id="4321"/>
    </pathologyReport>
  </sp:signedPayloadData>
</sp:signedPayload>

```

```

<patient id="5678"/>
<results>
  <test name="blood pressure" units="mmHg">90-119</test>
</results>
</pathologyReport>
</sp:signedPayloadData>
</sp:signedPayload>

```

2.3.3 Add the ID attribute

The `sp:signedPayloadData` element has an `id` attribute, which must have a unique value as required by criterion XS 3.1.1.1-3.

This ID value should be a globally unique identifier because the signed payload container could be composed into other XML documents that have `id` attributes. A globally unique identifier is generated using the `java.util.UUID` class.

```
String signId = UUID.randomUUID().toString();
```

An `id` attribute node is created, set to the generated globally unique identifier and added to the `sp:signedPayloadData` DOM element.

```
Attr idAttr = containerDoc.createAttributeNS(null, "id");
idAttr.setValue(signId);
signedPayloadDataElem.getAttributes().setNamedItem(idAttr);
```

After this step, the container document has the following XML:

```

<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
  <sp:signatures>
  </sp:signatures>
  <sp:signedPayloadData id="a0ede6c4-21e4-44d5-93b3-4b2f02f9e1f8">
    <pathologyReport xmlns="http://www.example.org">
      <creator id="1234"/>
      <receiver id="4321"/>
      <patient id="5678"/>
      <results>
        <test name="blood pressure" units="mmHg">90-119</test>
      </results>
    </pathologyReport>
  </sp:signedPayloadData>
</sp:signedPayload>

```

2.3.4 Normalize the document

The `normalizeDocument` method needs to be called on the container document before the signatures are created. This method transforms the DOM document to a normalized form. A part of the normalization process is to create XML attribute nodes for the declared XML namespaces. This is necessary to correctly calculate signature values.

```
containerDoc.normalizeDocument();
```

2.3.5 Get an XMLSignatureFactory instance

Most of the objects required by the signing process are provided by the `javax.xml.crypto.dsig.XMLSignatureFactory`. Depending on the way a program represents XML data, different types of `XMLSignatureFactory` are required. This example code is designed to process XML as DOM Documents.

```
XMLSignatureFactory xmlSigFactory = XMLSignatureFactory.getInstance("DOM");
```

2.3.6 Create the Reference object

The Reference object holds a representation of a `ds:Reference` element. The `XMLSignatureFactory`'s `newReference` method constructs a new

Reference object. It has 3 parameters: a URI, a digest method, and a list of transformations.

The URI value must be a fragment identifier referring to the XML element being signed, as required by criterion XS 4.4.3.1-3.

A `DigestMethod` object is created using the `newDigestMethod` method on the `XMLSignatureFactory` instance. The digest method must be set to SHA-1 as required by criterion XS 4.4.6.1-1.

Transform objects are created using the `newTransform` method on the `XMLSignatureFactory` instance. As required by criterion XS 4.4.5.1-1 and XS 4.4.5.1-2, there should only be one transform and this transform is for Exclusive XML Canonicalization.

```
String referenceUri = "#" + referenceId;

DigestMethod digestMethod = xmlSigFactory.newDigestMethod(DigestMethod.SHA1,
    null);

Transform transform = xmlSigFactory.newTransform(
    CanonicalizationMethod.EXCLUSIVE, (TransformParameterSpec) null);
List<Transform> transformList = Collections.singletonList(transform);

Reference reference = xmlSigFactory.newReference(referenceUri, digestMethod,
    transformList, null, null);
```

When serialized, this `Reference` object will appear as follows.

```
<ds:Reference URI="#a0ede6c4-21e4-44d5-93b3-4b2f02f9e1f8"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>Orn5miXZzb9EYhwcMg4ZTWavRds=</ds:DigestValue>
</ds:Reference>
```

2.3.7 Create the SignedInfo object

The `XMLSignatureFactory`'s `newSignedInfo` method constructs a new `SignedInfo` object. Its parameters are a canonicalization method, a signature method and a list of references.

The canonicalization method of the `SignedInfo` object must be Exclusive XML Canonicalization as required by criterion XS 4.4.1.1-1.

Criterion XS 4.4.2.1-1 requires the RSA-SHA1 as the signature method of the `SignedInfo` object.

The single `Reference` object that was previously constructed is passed within a `List`. This conforms to criterion XS 4.4.3.1-1 which only permits one `ds:Reference` element within a `ds:SignedInfo` element.

```
CanonicalizationMethod canonicalisationMethod = xmlSigFactory.
    newCanonicalizationMethod(CanonicalizationMethod.EXCLUSIVE,
    (C14NMethodParameterSpec) null);

SignatureMethod signatureMethod = xmlSigFactory.newSignatureMethod(
    SignatureMethod.RSA_SHA1, null);

List<Reference> referenceList = Collections.singletonList(reference);

SignedInfo signedInfo = xmlSigFactory.newSignedInfo(canonicalisationMethod,
    signatureMethod, referenceList);
```

When serialized, this `SignedInfo` object will appear as follows.

```
<ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:CanonicalizationMethod
    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
  <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
  <ds:Reference URI="#a0ede6c4-21e4-44d5-93b3-4b2f02f9e1f8">
    <ds:Transforms>
      <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
    </ds:Transforms>
  </ds:Reference>
</ds:SignedInfo>
```

```

</ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>Orn5miXZzb9EYhwcMg4ZTWavRds=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>

```

2.3.8 Create the Signatures

A signature is created for each pair of private key and X.509 certificate that was provided to the signing process.

```

for (CertificateKeyPair keyPair : keyPairs) {
    X509Certificate signingCertificate = keyPair.getCertificate();
    PrivateKey signingPrivateKey = keyPair.getPrivateKey();
}

```

2.3.8.1 Create a KeyInfo object

The XML signature must have a `ds:KeyInfo` element to conform to criterion XS 4.6.1.1-1. The `XMLSignatureFactory` provides a `KeyInfoFactory` for generating `KeyInfo` objects.

A `X509Data` object is first constructed using the `KeyInfoFactory`. The `X509Data` object must hold the X.509 certificate that was used to create the signature as per criterion XS 4.6.2.1-2.

This `X509Data` object is used to create the `KeyInfo` object. The `ds:KeyInfo` element will then have one `ds:X509Data` element as per criterion XS 4.6.2.1-1.

```

KeyInfoFactory keyinfoFactory = xmlSigFactory.getKeyInfoFactory();
X509Data x509Data = keyinfoFactory.newX509Data(
    Collections.singletonList(signingCertificate));
KeyInfo keyInfo = keyinfoFactory.newKeyInfo(
    Collections.singletonList(x509Data));

```

When serialized, this `KeyInfo` object will appear as follows.

```

<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:X509Data>
    <ds:X509Certificate>MIIC.....VGeirnEOmoPS2</ds:X509Certificate>
  </ds:X509Data>
</ds:KeyInfo>

```

2.3.8.2 Calculate the signature

The `XMLSignatureFactory`'s `newXMLSignature` method takes the generated `SignedInfo` object and `KeyInfo` object to create an `XMLSignature` object. A `DOMSignContext` is created from the `PrivateKey` and the signatures element passed in as arguments to the `signElement` method. The signature element will be the parent of the generated `Signature`. Finally, the `sign` method on the `XMLSignature` object is invoked to construct the signature, and insert it into the specified element.

Note that the `sign` method modifies the `signContext` argument by inserting the `Signature` element.

```

XMLSignature signature = xmlSigFactory.newXMLSignature(signedInfo, keyInfo);
DOMSignContext signContext = new DOMSignContext(
    signingPrivateKey, signaturesElement);

// Marshal and sign the signature elements
signature.sign(signContext);

```

When serialized, the `Signature` object will appear within the `sp:signatures` element. This destination is indicated by the `signContext`.

```

<sp:signedPayload
  xmlns:sp="http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0">
  <sp:signatures>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />

```

```

<ds:SignatureMethod
  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
<ds:Reference URI="#e08ea118-adff-4221-9f00-47e771a55261">
  <ds:Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
  </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <ds:DigestValue>xlihXCNm2/R/JldhDPhGwQrSb6c=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>...</ds:SignatureValue>
<ds:KeyInfo>
  <ds:X509Data>
    <ds:X509Certificate>...</ds:X509Certificate>
  </ds:X509Data>
</ds:KeyInfo>
</ds:Signature>
</sp:signatures>
<sp:signedPayloadData id="e08ea118-adff-4221-9f00-47e771a55261">
  <pathologyReport xmlns="http://www.example.org">
    <creator id="1234"/>
    <receiver id="4321"/>
    <patient id="5678"/>
    <results>
      <test name="blood pressure" units="mmHg">90-119</test>
    </results>
  </pathologyReport>
</sp:signedPayloadData>
</sp:signedPayload>

```

2.4 Verification Process

The steps involved in verifying a signed payload container that complies with the *XML Secured Payload Profile* [XSP2008] are:

- Getting the signature elements;
- Getting an XMLSignatureFactory instance;
- Checking the signatures; and
- Extracting the payload.

The inputs to the verification process are:

`XmlDocument containerDoc`

A `org.w3c.dom.Document` containing the signed payload container.

`KeyStore trustStore`

A store of trusted certificates.

The output of the verification process is:

A `org.w3c.dom.Document` containing the XML payload embedded in the signed payload container.

2.4.1 Get the signature elements

The `sp:signatures` element is retrieved from the root element of the container document since this element holds the digital signatures.

```

Element spSignaturesElem = XmlUtils.getChildElement(
  containerDoc.getDocumentElement(),
  "http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0",
  "signatures");

```

The `ds:Signature` elements are retrieved from the `sp:signatures` element.

```

List<Element> dsSignatureElems = XmlUtils.getChildElements(spSignaturesElem,
  "http://www.w3.org/2000/09/xmldsig#", "Signature");

```

The `getChildElements` method retrieves the children of a given element that has a specified namespace and name. The `getChildElement` method returns the first child that matches the namespace and name. The code for the `XmlUtils` class is given in appendix section D.2.

2.4.2 Get an XMLSignatureFactory instance

The objects required by the verification process are created using an `javax.xml.crypto.dsig.XMLSignatureFactory` instance. The `XMLSignatureFactory` instance must be DOM-based since this example code processes DOM documents.

```
XMLSignatureFactory factory = XMLSignatureFactory.getInstance("DOM");
```

2.4.3 Check the signatures

When there are multiple signatures, the security requirements of the context will determine which signatures need to be validated. In this example, every signature in the signed payload container is validated.

```
for (Element dsSignatureElem : dsSignatureElems) {
```

2.4.3.1 Unmarshal the ds:Signature element

The `ds:Signature` element must be unmarshalled using the `XMLSignatureFactory` instance into a `javax.xml.crypto.dsig.XMLSignature` object.

```
DOMStructure domStructure = new DOMStructure(dsSignatureElem);
XMLSignature signature = xmlSigFactory.unmarshalXMLSignature(domStructure);
```

2.4.3.2 Retrieve the signing certificate from the signature

The certificate of the private key that was used to create signature can be retrieved from the `KeyInfo` object of the XML signature.

```
KeyInfo keyInfo = signature.getKeyInfo();
X509Data x509Data = (X509Data) keyInfo.getContent().get(0);
X509Certificate certificate = (X509Certificate) x509Data.getContent().get(0);
```

2.4.3.3 Validating the signature

The validating context provides inputs to the signature validation. The signature to be validated and the public key to use to validate the signature must be provided to the validating context. A `javax.xml.crypto.dsig.dom.DOMValidateContext` instance is created as the validating context because the signature is represented in a DOM element.

```
DOMValidateContext valContext = new DOMValidateContext(
    certificate.getPublicKey(), dsSignatureElem);
```

Signature validation is done by passing the validating context to the `validate` method of the `XMLSignature` object. When the signed data has been changed, this method returns false. The verification process should take some action when a signature is invalid. The example code below throws an exception.

```
boolean valid = signature.validate(valContext);
if (!valid) {
    throw new XspException("Error: Signature verification failed.");
}
```

2.4.3.4 Verify the credentials

The signer's credentials are verified by checking the certificate of the private key used to sign the payload. This could involve checking the trust chain,

checking the certificate against a CRL, or invoking a separate service to perform these tasks. In this example, only the trust chain is checked.

The example code performs a trust chain check on the signing certificate. It attempts to build a certification path from the signing certificate to a CA certificate in the trust store. The trust store was provided as an input to the verification process. If a certification path cannot be established, then an exception will be thrown; the identity of the signing certificate cannot be trusted.

```
X509CertSelector selector = new X509CertSelector();
selector.setCertificate(signingCertificate);
PKIXBuilderParameters parameters = new PKIXBuilderParameters(trustStore,
    selector);
CertPathBuilder builder = CertPathBuilder.getInstance("PKIX");
builder.build(parameters);
```

2.4.4 Extract the payload

The payload will be the first child element of the `sp:signedPayloadData` element.

```
Element signedPayloadDataElem = XmlUtils.getChildElement(
    containerDoc.getDocumentElement(),
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/SignedPayload/1.0",
    "signedPayloadData");
Element payloadElem = XmlUtils.getFirstChildElement(signedPayloadDataElem);
```

The `getFirstChildElement` method returns the first child element within a given element. The `getChildElement` method returns the first child element that matches a given namespace and name. The code for the `XmlUtils` class is given in appendix section D.2.

The payload element must be imported into a separate DOM document.

```
DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
docBuilderFactory.setNamespaceAware(true);
DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
Document payloadDoc = docBuilder.newDocument();
Node nodeCopy = payloadDoc.importNode(payloadElem, true);
payloadDoc.appendChild(nodeCopy);
```

3 Encrypted Payload

The *XML Secured Payload Profile* [XSP2008] provides a mechanism called encrypted payload that uses *XML Encryption* [XENC2002] to ensure the confidentiality of XML data.

This chapter explains how to create and decrypt encrypted payloads that are conformant to the *XML Secured Payload Profile* [XSP2008].

Section 3.1 provides an introduction to the technologies and specifications relevant to encrypted payloads. It is a conceptual overview that is independent of implementations.

Apache XML Security has an API that implements *XML Encryption* that can be used to create and process encrypted payloads. Section 3.2 introduces this API and others used by the example code. Section 3.3 provides example code to encrypt XML data and create an encrypted payload. Section 3.4 provides example code that can decrypt an encrypted payload and extract the original XML data.

3.1 Conceptual overview

3.1.1 Encryption

Encryption is the process of transforming readable plaintext into unreadable cipher text. Decryption is the process of transforming the unreadable cipher text back into readable plain text. Encryption provides confidentiality between a sender and receiver as only the intended receiver is capable of decrypting and reading the data.

There are two basic schemes for encrypting data:

- Symmetric: A single secret key is shared between the sender and receiver. The secret key is used to encrypt and decrypt the data.
- Asymmetric: The keys come in pairs, where one is designated the private key and the other the public key. The sender uses the published public key of the receiver to encrypt the data. The receiver decrypts the data using the private key. Only the receiver has the private key.

A major issue with symmetric encryption is key management because it requires the secret key to be safely distributed to both the sender and receiver before data is sent. Asymmetric encryption solves the key management problem of symmetric encryption, requiring the sender to only obtain a publicly available key of the receiver to send data. However, asymmetric encryption is significantly slower than symmetric encryption. Both schemes are often combined in practice.

3.1.2 XML Encryption

Encryption can be performed on different types of data, not necessarily XML data. The *XML Encryption* specification [XENC2002] defines how encryption is applied to XML data. It specifies the processes for encrypting and decrypting XML data and the representation of the encryption result in XML.

Data is encrypted using XML Encryption by the following steps:

1. A random session key is generated.
2. The data is encrypted using a symmetric algorithm with the session key. Symmetric encryption is used for the data for better performance. The encrypted data is represented using the `xenc:EncryptedData` element.

3. The session key is encrypted using an asymmetric algorithm with the public key of the receiver. The encrypted session key is represented using the `xenc:EncryptedKey` element. The `xenc:EncryptedKey` element can use a `ds:KeyInfo` element to specify which key was used. The encrypted key can be added to the `ds:KeyInfo` element of the `xenc:EncryptedData` element or it can exist independently.

Data is decrypted using XML Encryption by the following steps:

1. The encrypted session key within the `xenc:EncryptedKey` element is decrypted using the private key of the receiver. The decrypted session key is the key that was used to encrypt the data.
2. The cipher text within the `xenc:EncryptedData` element is decrypted using the session key.

3.1.3 Encrypted payload container

The use of *XML Encryption* [XENC2002] doesn't guarantee that one implementation can decrypt XML data that is encrypted by another implementation since there are several options when using XML Encryption. Therefore, the *XML Secured Payload Profile* was created [XSP2008].

The *XML Secured Payload Profile* specification defines a way of using XML Encryption in order to enable interoperability when exchanging encrypted XML data. For instance, it specifies what algorithms to use, how data is referenced and how keys are referenced. It also defines how the encrypted data and keys are to be placed in a container XML document.

The example listing below shows an encrypted payload container conforming to the *XML Secured Payload Profile* specification [XSP2008].

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys>
    <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:X509Data>
          <ds:X509SKI>zdBo/qVF1VyLTi8VNoxSqs jU/B8=</ds:X509SKI>
        </ds:X509Data>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>
uQJemj3rJTgfm4hcPe5EebpoX306IBv9uLuOCSP6oKXYrFnPjCtI0A7423fkRC0dqbvMkN3xOtQ94yzhF4H
py2Fs6YIEf3Xf3r6s8I+huDecpEa4l0ZP4/+uVVL/FEmdZeRvf0ayhp0+miRD0rOtFP2peiNjt6G7ggIK5
vkOnNzw=
        </xenc:CipherValue>
      </xenc:CipherData>
      <xenc:ReferenceList>
        <xenc:DataReference URI="#_1" />
      </xenc:ReferenceList>
    </xenc:EncryptedKey>
  </ep:keys>
  <ep:encryptedPayloadData>
    <xenc:EncryptedData Id="_1"
      Type="http://www.w3.org/2001/04/xmlenc#Element"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />
      <xenc:CipherData>
        <xenc:CipherValue>
BQ6byRM5fjzm0IU1AYIHxd5ufsu7/ctJEGxRfo04JfGA4qa3PgTjaBry/9YN8wqblIoGxiNeyGAzenxONY
oky+AVMAdqtsh6QzKx1Ze3Xnj5I8oTHMA8EfL0R/w8ObuHhQxHnfxbf6yNIZik09dS6Z41pLAVyphEotaU
v+TWE+fdhds/ti3wnxqSSA8ESDIY8d6lN5P8A3AlZY73dynwWw8Ju8pWCtYQYzz+ezV1ng=
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </ep:encryptedPayloadData>
</ep:encryptedPayload>
```

3.2 Libraries used

The standard API for XML Encryption in Java, called *JSR-106 XML Digital Encryption APIs* [JSR106], is currently in draft form at the time of this document's publication. And also there are no known implementations of this draft API.

In the absence of an implementation of the standard XML Encryption API, this document uses the *Apache XML Security* [AXMLSEC] library to explain how to encrypt and decrypt XML data. The packages used from the *Apache XML Security* library are:

- org.apache.xml.security,
- org.apache.xml.security.encryption,
- org.apache.xml.security.keys and
- org.apache.xml.security.exceptions.

Classes from the general security libraries of the JDK are needed to load keys and perform cryptographic operations. The general security packages used are:

- java.security,
- java.security.cert, and
- javax.crypto.

XML data is represented and processed using the Document Object Model (DOM). The packages used for loading and processing DOM are:

- javax.xml.parsers, and
- org.w3c.dom.

The example code uses classes from the general packages of the JDK, namely:

- java.util, and
- java.io.

The following import statements apply to the code listings in sections 3.3 and 3.4:

```
import java.security.Key;
import java.security.PrivateKey;

import java.security.cert.X509Certificate;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

import org.apache.xml.security.Init;

import org.apache.xml.security.encryption.EncryptedData;
import org.apache.xml.security.encryption.EncryptedKey;
import org.apache.xml.security.encryption.Reference;
import org.apache.xml.security.encryption.ReferenceList;
import org.apache.xml.security.encryption.XMLCipher;
import org.apache.xml.security.encryption.XMLEncryptionException;

import org.apache.xml.security.keys.KeyInfo;
import org.apache.xml.security.keys.content.X509Data;
import org.apache.xml.security.keys.content.x509.XMLX509SKI;

import org.apache.xml.security.exceptions.XMLSecurityException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
```

```
import java.util.Arrays;
import java.util.List;
import java.util.UUID;

import java.io.StringWriter;
```

3.3 Encryption process

This section describes how to create an encrypted payload container that conforms to the *XML Secured Payload Profile* [XSP2008].

The process to create the encrypted payload container consists of:

- Creating the container document;
- Generating the session key;
- Generating the reference ID;
- Encrypting the data; and
- Encrypting the session key.

The input parameters to the encryption process are:

Document payloadDoc

A DOM Document containing the payload to be encrypted.

List<X509Certificate> certificates

A list of one or more X.509 certificates. These certificates are used to encrypt the session key. Only the matching private key for any of these certificates can be used to decrypt the encrypted document.

The output of the encryption process is:

A DOM Document structured according to the *EncryptedPayload* complex type in the *XML Secured Payload Schema* [XSP2008]

3.3.1 Create the container document

The *XML Secured Payload Profile* [XSP2008] specifies in criterion XS 3.1.2.1-1 an XML structure that acts as a container for the encrypted data and one or more encrypted keys. The section explains how to create this XML structure in a DOM document object.

The container document should be created first because many of the encryption operations require a context argument which must reference the DOM Document under construction.

3.3.1.1 Create an empty document

An empty DOM document is created by calling the `newDocument` method on an instance of the `javax.xml.parsers.DocumentBuilder` class.

For XML encryption to work correctly, the DOM document that is created must support XML namespaces. This is ensured by setting namespace aware to true on the factory that creates the `DocumentBuilder` instance.

```
DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
docBuilderFactory.setNamespaceAware(true);
DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();

Document containerDoc = docBuilder.newDocument();
```

3.3.1.2 Create the ep:encryptedPayload root element

The ep:encryptedPayload element is the root element of the container document.

The following creates the ep:encryptedPayload element and adds it as the root element to the container document:

```
Element encryptedPayloadElem = containerDoc.createElementNS(
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0",
    "ep:encryptedPayload");
containerDoc.appendChild(encryptedPayloadElem);
```

This results in the container document containing the following structure:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
</ep:encryptedPayload>
```

3.3.1.3 Create the ep:keys element

The ep:keys element is used to hold one or more xenc:EncryptedKey elements, which represents the encrypted session keys.

The following creates the ep:keys element and adds it as a child to the ep:encryptedPayload element:

```
Element keysElem = containerDoc.createElementNS(
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0",
    "ep:keys");
encryptedPayloadElem.appendChild(keysElem);
```

This results in the container document containing the following structure:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys/>
</ep:encryptedPayload>
```

3.3.1.4 Create the ep:encryptedPayloadData element

The ep:encryptedPayloadData element is used to hold the xenc:EncryptedData element, which represents the encrypted data.

The following creates the ep:encryptedPayloadData element and adds it as a child to the ep:encryptedPayload element:

```
Element encryptedPayloadDataElem = containerDoc.createElementNS(
    "http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0",
    "ep:encryptedPayloadData");
encryptedPayloadElem.appendChild(encryptedPayloadDataElem);
```

This results in the container document containing the following structure:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys/>
  <ep:encryptedPayloadData/>
</ep:encryptedPayload>
```

3.3.2 Generate the session key

Criterion XS 5.8.1.1-6 requires that a new session key be generated every time a payload is encrypted. The generation of the session key depends on the symmetric algorithm that will be used to encrypt the data. Criterion XS 5.4.1.1-2 requires the use of the AES-256 algorithm for data encryption.

The session key can be generated in Java using the javax.crypto.KeyGenerator class. AES is specified when getting a KeyGenerator instance. This KeyGenerator instance must be initialised with the key size of 256 before the key is generated.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init(256);
```

```
SecretKey sessionKey = keyGenerator.generateKey();
```

3.3.3 Generate the reference ID

The `xenc:EncryptedData` element must be given an `Id` attribute that is set to a unique value as per criterion XS 5.3.1.1-2. This ID value is referenced by the `xenc:EncryptedKey` elements to link the encrypted keys with the encrypted data. This ID value must be globally unique as the container could be composed within other containers, or the payload itself could have IDs.

The `java.util.UUID` class can be used to generate a globally unique identifier in Java. An underscore is prefixed to the UUID as the *XML Encryption XML Schema [XENC2002]* specifies an ID (NCName) data type for the `Id` attribute. An NCName value must start with either a letter or an underscore.

```
String referenceId = "_" + UUID.randomUUID().toString();
```

3.3.4 Encrypt the data

3.3.4.1 Create an XMLCipher

The encryption is performed using the `org.apache.xml.security.encryption.XMLCipher` class. The data encryption algorithm is specified when getting an `XMLCipher` instance. This must be AES-256 algorithm as per criterion XS 5.4.1.1-2. The `XMLCipher` instance must be initialised with the key to encrypt the data, which should be the session key generated in section 3.3.2.

```
XMLCipher dataCipher = XMLCipher.getInstance(XMLCipher.AES_256);
dataCipher.init(XMLCipher.ENCRYPT_MODE, sessionKey);
```

3.3.4.2 Create the EncryptedData object

The data is encrypted by providing the container document created in section 3.3.1 and the root element of the payload document to be encrypted.

```
EncryptedData encryptedData =
    dataCipher.encryptData(containerDoc, payloadDoc.getDocumentElement());
```

3.3.4.3 Set the Id attribute to EncryptedData

The reference ID generated in section 3.3.3 is set as the ID of `EncryptedData` object. This step will ensure that the `xenc:EncryptedData` element has an `Id` attribute as per criterion XS 5.3.1.1-2.

```
encryptedData.setId(referenceId);
```

3.3.4.4 Add to ep:encryptedPayloadData element

The `EncryptedData` object must be converted to a DOM Element and added as a child of the `ep:encryptedPayloadData` element.

```
Element encryptedDataElem = dataCipher.martial(encryptedData);
encryptedPayloadDataElem.appendChild(encryptedDataElem);
```

After these steps, the container document has the following XML:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys />
  <ep:encryptedPayloadData>
    <xenc:EncryptedData Id="_8a11b570-5551-4d98-9004-877e5dc4b0d2"
      Type="http://www.w3.org/2001/04/xmlenc#Element"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />
      <xenc:CipherData>
        <xenc:CipherValue>
```

```
BQ6byRM5fjzm0IU1AYIHxd5ufsu7/ctJEGxRfo04JfGA4qa3PgTjaBry/9YN8wqb1IoGxiNeyGAzenxONY
oky+AVMAdqtsh6QzKx1Ze3Xnj5I8oTHMA8EfLOR/w8ObuHhQxHnfxbf6yNIZik09dS6Z41pLAVyphEotaU
v+TWE+fdhds/ti3wnxqSSA8EsDIY8d61N5P8A3AlZY73dynwWw8gJu8pWCtYQYzz+ezV1ng=
  </xenc:CipherValue>
  </xenc:CipherData>
</xenc:EncryptedData>
</ep:encryptedPayloadData>
</ep:encryptedPayload>
```

3.3.5 Encrypt session key for each receiver

A payload may be encrypted for one or more receivers. For every receiver, the session key must be asymmetrically encrypted using the RSA 1.5 algorithm (criteria XS 5.7.2.1-2) and the public key of the receiver (criteria XS 5.7.4.1-1).

The code snippets in this section must be executed using the public certificates of each receiver to produce one or more `EncryptedKey(s)`. Each `EncryptedKey` will contain the encrypted session key, details of the public certificate used to encrypt the session key, and a reference to the encrypted data.

3.3.5.1 Create `EncryptedKey`

The following code creates an `XMLCipher`, initialised for key wrapping using the public key of the certificate, and encrypts the session key:

```
XMLCipher keyCipher = XMLCipher.getInstance(XMLCipher.RSA_v1dot5);
keyCipher.init(XMLCipher.WRAP_MODE, certificate.getPublicKey());
EncryptedKey encryptedKey = keyCipher.encryptKey(containerDoc, sessionKey);
```

3.3.5.2 Create `KeyInfo` and add to `EncryptedKey`

Each `EncryptedKey` must contain `KeyInfo` (criteria XS 5.7.3.1-1).

```
KeyInfo keyInfo = new KeyInfo(containerDoc);
encryptedKey.setKeyInfo(keyInfo);
```

3.3.5.3 Create `X509Data` and add to `KeyInfo`

Each `KeyInfo` must contain a single `X509Data` (criteria XS 5.7.4.1-2).

```
X509Data x509Data = new X509Data(containerDoc);
keyInfo.add(x509Data);
```

3.3.5.4 Create SKI and add to `X509Data`

Each `X509Data` must contain the Subject Key identifier (SKI) of the public certificate (criteria XS 5.7.4.1-3).

```
x509Data.addSKI(certificate);
```

3.3.5.5 Create `ReferenceList` and add to `EncryptedKey`

Each `EncryptedKey` must contain a `ReferenceList` and have a single `DataReference` with the URI attribute set to the Id attribute of the encrypted data (criteria XS 5.7.1.1-4).

```
ReferenceList referenceList =
keyCipher.createReferenceList(ReferenceList.DATA_REFERENCE);
encryptedKey.setReferenceList(referenceList);
Reference dataReference = referenceList.newDataReference("#" + referenceId);
referenceList.add(dataReference);
```

3.3.5.6 Create `EncryptedKey` element

Each `EncryptedKey` object must be converted to a DOM Element and added as a child of the keys element.

```
Element encryptedKeyElem = keyCipher.martial(encryptedKey);
```

```
keysElem.appendChild(encryptedKeyElem);
```

An example of the partial container Document after this stage is given below:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys>
    <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
    />
      <xenc:KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <xenc:X509Data>
          <xenc:X509SKI>pQOZFLO6xwkfMGRF+zW+Jt5Nczs=</xenc:X509SKI>
        </xenc:X509Data>
      </xenc:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>
uQJemj3rJTgfm4hcPe5EebpoX3O6IBv9uLuOCSP6oKXYrFnPjC10A7423fkRC0dgbvMkN3xOtQ94yzhF4H
py2Fs6YIEf3Xf3r6s8I+huDecpEa410ZP4/+uVVL/FEmdZeRvf0ayhp0+miRD0rOtFP2peiNJt6G7ggIK5
vkOnNzw=
        </xenc:CipherValue>
      </xenc:CipherData>
      <xenc:ReferenceList>
        <xenc:DataReference URI="#_8a11b570-5551-4d98-9004-877e5dc4b0d2" />
      </xenc:ReferenceList>
    </xenc:EncryptedKey>
  </ep:keys>
  <ep:encryptedPayloadData/>
</ep:encryptedPayload>
```

3.4 Decryption process

This section describes how to process an encrypted payload container that conforms to the *XML Secured Payload Profile* [XSP2008].

The process to process the encrypted payload consists of:

- Finding the encrypted key;
- Decrypting the encrypted key;
- Decrypting the payload; and
- Extracting the payload.

The input parameter to the decryption process is:

Document containerDoc

A DOM Document containing a structure conforming to the EncryptedPayload XML Schema [XSP2008].

PrivateKey decryptKey

The private key of the receiver.

X509Certificate decryptCert

The public X.509 certificate which matches the private key.

The output of the decryption process is:

A DOM Document containing the decrypted payload.

3.4.1 Find the encrypted key

The encrypted payload may have been encrypted for more than one receiver. It is necessary to search for the `xenc:EncryptedKey` within the `ep:EncryptedPayload` that matches the key pair of the receiver.

3.4.1.1 Find the `xenc:EncryptedKey` elements

The `xenc:EncryptedKey` elements are found at the path `/ep:keys/xenc:EncryptedKey`.

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys>
    <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      ...
    </xenc:EncryptedKey>
  </ep:keys>
  ...
</ep:encryptedPayload>
```

To obtain the list of `xenc:EncryptedKey` elements:

```
Element encryptedPayloadElem = containerDoc.getDocumentElement();
Element keysElem = XmlUtils.getChildElement(encryptedPayloadElem,
  "http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0",
  "keys");
List<Element> encryptedKeyElems = XmlUtils.getChildElements(keysElem,
  "http://www.w3.org/2001/04/xmlenc#", "EncryptedKey");
```

The `getChildElements` method retrieves the children of a given element that has a specified namespace and name. The `getChildElement` method returns the first child that matches the namespace and name. The code for the `XmlUtils` class is given in appendix section D.2.

The operations below in sections 3.4.1.2 and 3.4.1.3 must be performed for each `EncryptedKey` until a match is found with the key pair of the receiver.

```
for (Element encryptedKeyElem : encryptedKeyElems) {
```

3.4.1.2 Unmarshal `EncryptedKey` element

To unmarshal an `xenc:EncryptedKey` DOM Element to an `EncryptedKey` object:

```
XMLCipher xmlCipher = XMLCipher.getInstance();
xmlCipher.init(XMLCipher.DECRYPT_MODE, null);

EncryptedKey encryptedKey = xmlCipher.loadEncryptedKey(encryptedKeyElem);
```

3.4.1.3 Check for matching certificate

As per the encryption process, each `EncryptedKey` contains the Subject Key Identifier (SKI) of the X.509 certificate used to encrypt the session key.

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  <ep:keys>
    <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
      <xenc:KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <xenc:X509Data>
          <xenc:X509SKI>pQOZFLO6xwkmGRF+zW+Jt5Nczs=</xenc:X509SKI>
        </xenc:X509Data>
      </xenc:KeyInfo>
      ...
      <xenc:ReferenceList>
        <xenc:DataReference URI="#_8a11b570-5551-4d98-9004-877e5dc4b0d2" />
      </xenc:ReferenceList>
    </xenc:EncryptedKey>
  </ep:keys>
  ...
</ep:encryptedPayload>
```

An `EncryptedKey` matches the receiver's key pair if the SKI of the `EncryptedKey` equals the SKI of the receiver's X.509 certificate.

```
KeyInfo keyInfo = encryptedKey.getKeyInfo();
X509Data x509Data = keyInfo.itemX509Data(0);
byte[] x509DataSki = x509Data.itemSKI(0).getSKIBytes();
```

```
byte[] decryptCertSki = XMLX509SKI.getSKIBytesFromCert(decryptCert);
boolean skiMatches = Arrays.equals(x509DataSki, decryptCertSki);
```

3.4.2 Find encrypted data

The encrypted data is located at the path

`ep:encryptedPayloadData/xenc:EncryptedData` within the encrypted payload.

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  ...
  <ep:encryptedPayloadData>
    <xenc:EncryptedData Id="_8a11b570-5551-4d98-9004-877e5dc4b0d2"
      Type="http://www.w3.org/2001/04/xmlenc#Element"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      ...
    </xenc:EncryptedData>
  </ep:encryptedPayloadData>
</ep:encryptedPayload>
```

To obtain the `xenc:EncryptedData` element:

```
Element encryptedPayloadDataElem = XmlUtils.getChildElement(encryptedPayloadElem,
  "http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0",
  "encryptedPayloadData");
Element encryptedDataElem = XmlUtils.getChildElement(encryptedPayloadDataElem,
  "http://www.w3.org/2001/04/xmlenc#", "EncryptedData");
```

3.4.3 Determine data encryption algorithm

The AES-256 algorithm must be used to encrypt the payload.

```
<ep:encryptedPayload
  xmlns:ep="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0">
  ...
  <ep:encryptedPayloadData>
    <xenc:EncryptedData Id="_8a11b570-5551-4d98-9004-877e5dc4b0d2"
      Type="http://www.w3.org/2001/04/xmlenc#Element"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:encryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-
        cbc" />
      ...
    </xenc:EncryptedData>
  </ep:encryptedPayloadData>
</ep:encryptedPayload>
```

For verification purposes, it is possible to extract the encryption algorithm from the `EncryptedData` either using DOM methods or using the XML Security API as follows:

```
EncryptedData encryptedData = xmlCipher.loadEncryptedData(
  containerDoc, encryptedDataElem);
String encryptionAlgorithm = encryptedData.getEncryptionMethod().getAlgorithm();
```

3.4.4 Decrypt session key

The session key can be decrypted from the `EncryptedKey` using the private key of the receiver:

```
xmlCipher.init(XMLCipher.DECRYPT_MODE, decryptKey);
Key sessionKey = xmlCipher.decryptKey(encryptedKey, encryptionAlgorithm);
```

3.4.5 Decrypt payload

Finally, the payload can be decrypted from the `EncryptedData` using the session key:

```
xmlCipher.init(XMLCipher.DECRYPT_MODE, sessionKey);
Document payloadDoc = xmlCipher.doFinal(
  containerDoc, encryptedDataElem);
```

4 Composite Payload

4.1 Conceptual Overview

The secured containers can be composed within each other. For example, a signed payload container could be the payload of an encrypted payload container.

The following are common patterns for composing the containers:

- Signed Encrypted.

These will be described in the following sections.

4.2 Signed Encrypted Payload

A signed-encrypted payload is created by:

1. Signing the payload as per the signing process in section 2.3;
2. Using the signed payload as input to the encryption process, specified in section 3.3.

5 Secured payloads with Web services

It is expected that one of the most common uses for an XML secured payload is as an argument within the SOAP body of a Web service request. Web service interfaces are defined using WSDL and XML Schema, with the complex data type defined in the XML Schema.

The recommended approach for implementing Web services in Java is to use the JAX-WS/JAXB frameworks. These frameworks attempt to generate concrete types (specific Java classes) that correspond to each of the complex types used by the Web services. The example Java code defined in this document - and more relevantly the *JSR-105 [JSR105]* and *Apache XML Security APIs [AXMLSEC]* - use DOM Document and Element objects as arguments. It is therefore better that the JAX-WS/JAXB frameworks receive/provide XML secured payloads as DOM objects rather than complex EncryptedPayload and/or SignedPayload objects.

There are two recommended ways that a secured payload can be implemented using JAXB:

1. Using an XML Schema "any" declaration;
2. Using an XML Schema "element" declaration and appropriate JAXB bindings to manipulate the generated classes.

5.1 Secured payload as XML Schema "any"

The simplest way to declare an XML secured payload with XML Schema is to define the contents using the XML Schema "any" declaration.

```
<xsd:complexType name="SecuredReportAny">
  <xsd:sequence>
    <xsd:element name="reportId" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="receiver" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="sender" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
    <xsd:any processContents="lax"
namespace="http://ns.nehta.gov.au/CoreConnectivity/Xsd/EncryptedPayload/1.0"
minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

The Java class generated for this XML Schema by `wsimport` or `xjc` will resemble:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SecuredReportAny", propOrder = {
    "reportId",
    "sender",
    "receiver",
    "any"
})
public class SecuredReportAny {

    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String reportId;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String sender;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String receiver;
    @XmlAnyElement(lax = true)
    protected Object any;

    <code removed>
```

```

/**
 * Gets the value of the any property.
 *
 * @return
 *     possible object is
 *     {@link Element }
 *     {@link Object }
 *
 */
public Object getAny() {
    return any;
}

/**
 * Sets the value of the any property.
 *
 * @param value
 *     allowed object is
 *     {@link Element }
 *     {@link Object }
 *
 */
public void setAny(Object value) {
    this.any = value;
}

```

The any variable declared as a Java Object with corresponding getter and setter. When used within a Web service JAX-WS context, the actual type of object will be a DOM Element.

5.2 Secured payload as XML Schema “element”

A more complex but ultimately more user friendly way to declare an XML secured payload with XML Schema is to define the contents using the XML Schema “element” declaration using the appropriate complex type from the XML Secured Payload Schema.

```

<xsd:complexType name="SecuredReportElement">
  <xsd:sequence>
    <xsd:element name="reportId" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="receiver" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="sender" type="xsd:anyURI" minOccurs="1" maxOccurs="1"/>
    <xsd:element ref="ep:encryptedPayload" minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

```

As mentioned at the start of this chapter, it is preferable to implement the securedPayload variable as a DOM Element, not the complex EncryptedPayload type. To achieve this in JAXB, a binding such as this must be used when generating the Java class using wsimport or xjc:

```

<jaxb:bindings schemaLocation="../../../schemas/SecuredReport.xsd" node="/xs:schema">
  <jaxb:bindings
node="//xs:complexType[@name='SecuredReportElement']//xs:element[@ref='xep:encryptedPayload']">
    <jaxb:dom/>
  </jaxb:bindings>
</jaxb:bindings>

```

This instructs the JAXB code generator to use a DOM Element for the securedPayload variable.

The Java class generated for this XML Schema by wsimport or xjc will resemble:

```

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SecuredReportElement", propOrder = {
    "reportId",
    "sender",
    "receiver",
    "encryptedPayload"
})

```

```
public class SecuredReportElement {

    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String reportID;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String sender;
    @XmlElement(required = true)
    @XmlSchemaType(name = "anyURI")
    protected String receiver;
    @XmlAnyElement
    protected Element encryptedPayload;

    <code removed>

    /**
     * Gets the value of the encryptedPayload property.
     *
     * @return
     *     possible object is
     *     {@link Element }
     */
    public Element getEncryptedPayload() {
        return encryptedPayload;
    }

    /**
     * Sets the value of the encryptedPayload property.
     *
     * @param value
     *     allowed object is
     *     {@link Element }
     */
    public void setEncryptedPayload(Element value) {
        this.encryptedPayload = value;
    }

}
```

In this case the `encryptedPayload` element was implemented as an `Element` variable with corresponding getter and setter. This is an improvement both in terms of the name and type of the variable.

Appendix A: References

- [AXMLSEC] *Apache XML Security*, Release 1.4.2, Apache Software Foundation, June 2008.
<<http://xml.apache.org/security>>
- [EIJAX2008] NEHTA, *Example Technical Implementation of Interoperable Web Services: JAX-WS v3.0*, 1 December 2008.
- [EJBCA] EJBCA, *EJBCA – The Java EE Certificate Authority*,
<http://ejbca.sourceforge.net>.
- [JCA] Sun Microsystems, *Java Cryptography Architecture (JCA)*,
<http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>.
- [JDK6] Java Development Kit, Release 6, Sun Microsystems.
<<http://java.sun.com/javase/downloads>>
- [JSR105] *JSR-105 XML Digital Signature APIs*, 1.0 Final Release, Java Community Process, June 2005.
<<http://jcp.org/aboutJava/communityprocess/final/jsr105>>
- [JSR106] *JSR-106 XML Digital Encryption APIs*, Public Review Draft, Java Community Process, January 2006.
<<http://jcp.org/aboutJava/communityprocess/pr/jsr106>>
- [KEYTOOL] Sun Microsystems, *keytool - Key and Certificate Management Tool*,
<http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html>.
- [METRO] *Metro Web Services Stack* (online), Release 1.3, Sun Microsystems, August 2008.
<<https://metro.dev.java.net>>
- [OPENSSL] OpenSSL, *OpenSSL: The Open Source toolkit for SSL/TLS*,
<http://www.openssl.org>.
- [PKCS1999] RSA Laboratories, *PKCS 12: Personal Information Exchange Syntax*, version 1.0, 24 June 1999,
<ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf>.
- [WSP2008] NEHTA, *Web Services Profile v3.0*, 1 December 2008.
- [XENC2002] *XML Encryption Syntax and Processing*, W3C Recommendation XML Schema, W3C, 10 December 2002.
<<http://www.w3.org/TR/xmlenc-core/xenc-schema.xsd>>
- [XSP2008] NEHTA, *XML Secured Payload Profile v1.0*, 1 December 2008.

Appendix B: Key management

The term, *key store*, has 2 meanings in Java depending on the context. The first meaning refers to files containing security tokens, such as certificates and private keys. It is used in relation to general security tools and APIs. The second meaning is more specific, referring to the file containing an entity's own certificate and private key. It is differentiated from a *trust store*, which refers to the file containing other entities' public certificates. It is used in relation to Web service and SSL (Secure Sockets Layer) APIs. This appendix uses the first meaning.

B.1 Key store types

The standard Java distribution supports 3 key store types:

- PKCS #12
 - PKCS #12 belongs to the Public-Key Cryptography Standards (PKCS) group of specifications developed by RSA Laboratories. PKCS #12 is a standard format for storing and transferring identity information, such as certificates and private keys [PKCS1999].
 - The standard name for this key store type is *pkcs12*.
- Java Key Store (JKS)
 - JKS is a proprietary Java format for storing security tokens [JCA].
 - The standard name for this key store type is *jks*.
- Java Cryptography Extension Key Store (JCEKS)
 - JCEKS is another proprietary Java format, but it has stronger protection for private keys than JKS [JCA].
 - The standard name for this key store type is *jceks*.

Other key store types can be supported in Java using the extensible mechanisms of the Java Cryptography Architecture (JCA) [JCA].

B.2 Tools

The Java Development Kit (JDK) comes with a *keytool* command-line tool for managing keys and certificates [KEYTOOL]. This tool is found in the `<JDK_HOME>/bin` directory. It allows you to create key stores, import certificates into the key stores, list the keys in a key store, create self-signed certificates and more. Read the documentation for the *keytool* tool for the installed JDK. The functionality and command-line arguments for the tool can differ between JDK versions.

There are also open source tools, like openSSL [OPENSSL] and EJBCA [EJBCA], which allows you to create keys and certificates.

Appendix C: Installation

This appendix describes the installation process for the software and related components that are required to run the software examples described in this document.

C.1 Java Development Kit

1. Download and install JDK 6 Update 10 or later.

URL: <http://java.sun.com/javase/downloads/index.jsp>

<JDK_HOME> will be used in this document to refer to the root directory of the JDK installation.

<JRE_HOME> will be used in this document to refer to <JDK_HOME>/jre.

2. Add <JDK_HOME>/bin to the path.
3. Create a JAVA_HOME environment variable pointing to <JDK_HOME>.

C.2 Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files

The Java Cryptography Extension (JCE) provides cryptography services in the JDK. The JCE policy files in the JDK download are limited in strength due to the import control restrictions for some countries. The “unlimited strength” capabilities are enabled by installing certain policy files into the JRE.

4. Download the JCE Unlimited Strength Jurisdiction Policy Files for the installed JDK version.

URL: <http://java.sun.com/javase/downloads/index.jsp>

5. Unpack the downloaded ZIP file.
6. Copy the two JAR files (*local_policy.jar* and *US_export_policy.jar*) to the <JRE_HOME>/lib/security directory.

Overwrite the existing JAR files in the directory.

C.3 Apache XML Security API

1. Download the *Apache XML Security API 1.4.2* or later

URL: <http://xml.apache.org/security>

2. Unpack the downloaded ZIP file.

<AXMLSEC_HOME> will be used in this document to refer to the root directory of the Apache XML Security API distribution.

3. Include the JAR files in <AXMLSEC_HOME>/libs in the CLASSPATH of any Java application using the Apache XML Security encryption APIs.

Appendix D: Helper classes

D.1 CertificateKeyPair

The `CertificateKeyPair` class is a simple data structure to hold a `java.security.cert.X509Certificate` and its corresponding `java.security.PrivateKey`.

```
import java.security.PrivateKey;
import java.security.cert.X509Certificate;

public class CertificateKeyPair {

    private X509Certificate certificate;

    private PrivateKey privateKey;

    public X509Certificate getCertificate() {
        return this.certificate;
    }

    public void setCertificate(X509Certificate certificate) {
        this.certificate = certificate;
    }

    public PrivateKey getPrivateKey() {
        return this.privateKey;
    }

    public void setPrivateKey(PrivateKey privateKey) {
        this.privateKey = privateKey;
    }

}
```

D.2 XmlUtils

The `XmlUtils` class contains commonly used XML functions, such as processing DOM nodes.

```
import java.util.ArrayList;
import java.util.List;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public final class XmlUtils {

    public static Element getFirstChildElement(Node node) {
        NodeList childNodes = node.getChildNodes();
        if (childNodes.getLength() > 0) {
            // Iterate through each child node looking for an element
            for (int i = 0; i < childNodes.getLength(); i++) {
                Node currentNode = childNodes.item(i);
                if (currentNode instanceof Element) {
                    return (Element) currentNode;
                }
            }
        }
        return null;
    }

    public static Element getChildElement(Element element, String namespace,
        String tagName) {
        List<Element> childElems = getChildElements(element, namespace, tagName);
        if (childElems.size() == 0) {
            throw new IllegalArgumentException("No '{" + namespace + "'" + tagName
                + "' element found.");
        } else if (childElems.size() > 1) {
            throw new IllegalArgumentException("More than one '{" + namespace + "'"
                + tagName + "' elements found.");
        }
        return childElems.get(0);
    }

}
```

```
public static List<Element> getChildElements(Element element,
    String namespace, String tagName) {
    List<Element> childElems = new ArrayList<Element>();
    NodeList childNodes = element.getChildNodes();
    for (int i = 0; i < childNodes.getLength(); i++) {
        Node childNode = childNodes.item(i);
        if (childNode instanceof Element) {
            Element childElem = (Element) childNode;
            String currNamespace = childElem.getNamespaceURI();
            String currLocalName = childElem.getLocalName();
            if (namespace.equals(currNamespace) && tagName.equals(currLocalName)) {
                childElems.add(childElem);
            }
        }
    }
    return childElems;
}
```