

nehta

Example Technical Implementation of the XML Secured Payload Profile

Microsoft .NET

Version 1.1 — 30 June 2010

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 © 2010, 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

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	6
2.3.1	Create the container document	7
2.3.2	Import the payload.....	8
2.3.3	Add the ID attribute	8
2.3.4	Create the signatures	8
2.4	Verification process	11
2.4.1	Get the signature elements.....	12
2.4.2	Check the signatures	12
2.4.3	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.....	17
3.3.1	Create the container document	18
3.3.2	Import the payload.....	19
3.3.3	Generate the session key	19
3.3.4	Generate the reference ID	19
3.3.5	Encrypt the data	20
3.3.6	Encrypt the session key	20
3.4	Decryption process.....	22
3.4.1	Find the encrypted key.....	22
3.4.2	Decrypt the encrypted key.....	23
3.4.3	Decrypt the payload	23
3.4.4	Extract the payload	24
4	Composite Payload	25
4.1	Conceptual Overview	25
4.2	Signed Encrypted Payload	25
5	Secured payloads with Web services	26
	Appendix A: References.....	28
	Appendix B: Key management.....	29
B.1	Setting up the certificate MMC	29

B.2	Installing certificates	29
B.2.1	Add a certificate with a private key	29
B.2.2	Add a certificate.....	29
B.2.3	Adding a CA certificate.....	30
B.3	Accessing the store from code	30

Document information

Change history

Version	Date	Comments
1.0	2008-12-01	Release
1.1	2010-06-30	Use ATS 5821—2010 XSP profile.

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 communication between organisations 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* [ATS 5821—2010] 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 the XML Security API that is a part of the Microsoft .NET framework.

The main purpose of this document is to support the understanding and interpretation of the criteria in the *XML Secured Payload Profile* [ATS 5821—2010]. 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 an implementation in .NET. Also available is an example technical implementation document covering Java. 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 C#, 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* [ATS 5821—2010]. The criteria from [ATS 5821—2010] 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
CRL	Certificate Revocation List
DOM	Document Object Model
GUID	Globally Unique Identifier
MMC	Microsoft Management Console
OAEP	Optimal Asymmetric Encryption Padding
SKI	Subject Key Identifier
XML	Extensible Mark-up Language

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.electronichealth.net.au/xsp/xsd/SignedPayload/2010 (Signed payload [ATS 5821–2010])
ep	http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010 (Encrypted payload [ATS 5821–2010])
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, namespaces • Code fragments, names of classes, methods, and fields • Assemblies, packages

	<ul style="list-style-type: none">• 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 describes how to use secured payloads with Web services.

Appendix A: lists references.

Appendix B: describes key management in Windows.

2 Signed Payload

The *XML Secured Payload Profile* [ATS 5821—2010] 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* [ATS 5821—2010].

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 Microsoft .NET framework has an API that implements *XML Signature*, which can be used to create and process signed payloads. Section 2.2 introduces the .NET XML Signature API and other 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 validate 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:

1. 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.
2. 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 [ATS 5821—2010].

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 [ATS 5821—2010].

```
<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
  <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
FY0DZCf9xTEw2HTOCSRvRpxlL0DPbVc5JylUOv3e1hbcglCmhqfRyI9003b3se9WpF/nkDSZGAov2NmhWn
bnNcqjE=
      </ds:SignatureValue>
      <ds:KeyInfo>
        <ds:X509Data>
<ds:X509Certificate>MIICBjCCAW+gAwIBAgIBBjANBgkqhkiG9w0BAQUFADAeMRwwGgYDVQQDExNORU
hUQSBFSUlXUyBkZWlvIENBMB4XDTA3MDYyMjAwMDI1NFoXDTE3MDYyOTAwMDI1NFowFzEVMBMGA1UEAxQM
amF4d3NfY2xpZW50MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDVntVihAhN76KhBevo/YTzBxloB7
K8YqRhfG3ca/Y9qFv1Mk6tUNjC9LxBqtLcQbcLpeZOfsKumtygvvZUBAZlpTR7qMOeHcFMYCPlsVS4mTo
TRt8P33au2x6VvTj7AWaXr8Di1jv13hB/luPKRuflyw0hV7mzvSkc1wHVBBnQIDAQABo1swWTAMBGNVHR
MBAf8EAjAAMB0GA1UdDgQWBBSlA5kUs7rHCR8wZEX7Nb4m3klzOzAfBgNVHSMEGDAWgBQL0DcCBAjrsOSF5
Usiwo070lHtHBzAJBgNVHREEAjAAMA0GCSqGSIb3DQEBAQUAA4GBAGEP1ku0g5RftVWrfP+PSgHueUugWH
hOqUsUB5w30PYhkIawVfbrXu4cQ+wSo96yiP/89xsxTmMwiWa0LQ02xmeZWF2F9FrcO2Ni9nZ1lulREtrd
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

The Microsoft .NET framework contains an API that implements the *XML Signatures* specification [XDSIG2002]. This document describes how to create and validate XML signatures using that API.

The XML Signature API in the .NET framework is accessed by adding the `System.Security` assembly to a project. Its classes belong to the .NET namespace, `System.Security.Cryptography.Xml`. The `System.Security.Cryptography.Xml.SignedXml` class is the main class used to create and validate signatures.

The keys used to sign data can be retrieved from the Windows certificate repository. They can be accessed using the `System.Security.Cryptography.X509Certificates.X509Store` class.

The following .NET namespaces apply to the code listings in sections 2.3 and 2.4:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Xml;
using System.Security.Cryptography.Xml;
using System.Security.Cryptography.X509Certificates;
using System.Security.Cryptography;
using System.IO;
using System.Xml.Serialization;
```

2.3 Signing process

The steps involved in creating a signed payload container that complies with the *XML Secured Payload Profile* [ATS 5821–2010] are:

- Creating the container document;
- Importing the payload into the container;
- Adding the ID attribute onto the payload; and
- Creating the signatures over the identified payload.

The inputs to creating the signed payload container are:

`XmlDocument payloadDoc`

A DOM `XmlDocument` containing the payload to be signed. This can be any XML document.

`List<X509Certificate2> signCertificates`

A list of one or more X.509 certificates with their corresponding private key. The private keys are used to sign the payload. They can be retrieved from the Windows certificate repository. See appendix B.3 for details on retrieving the certificates and private keys.

The output of creating the signed payload container is:

A DOM `XmlDocument` containing XML structured according to the signed payload XML Schema defined in *XML Secured Payload Profile [ATS 5821—2010]*.

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 [ATS 5821—2010]* 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 an `XmlDocument` object.

2.3.1.1 Create an empty document

The container document holds the signatures and the signed data. This is created as an `XmlDocument`. It is important that the `PreserveWhitespace` property is set to `true` to ensure that the payload content is not modified when the container document is serialized to XML. Otherwise, the removal of whitespaces will cause signature validation failures.

```
XmlDocument containerDoc = new XmlDocument();
containerDoc.PreserveWhitespace = true;
```

2.3.1.2 Create the sp:signedPayload element

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

```
XmlElement signedPayloadElem = containerDoc.CreateElement(
    "sp", "signedPayload",
    "http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010");
containerDoc.AppendChild(signedPayloadElem);
```

This results in the container document containing the following structure:

```
<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
</sp:signedPayload>
```

2.3.1.3 Create the sp:signatures element

The `sp:signatures` element is created and added to the `sp:signedPayload` root element. The `sp:signatures` element is used to hold the XML signatures. Section 2.3.4 describes how to create the XML signatures and add them to this element.

```
XmlElement signaturesElem = containerDoc.CreateElement(
    "sp", "signatures",
    "http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010");
signedPayloadElem.AppendChild(signaturesElem);
```

This results in the container document containing the following structure:

```
<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
  <sp:signatures />
</sp:signedPayload>
```

2.3.1.4 Create the sp:signedPayloadData element

The `sp:signedPayloadData` element is also created and added to the `sp:signedPayload` root element. The `sp:signedPayloadData` element is used to hold the payload data being signed. Section 2.3.2 how to import the payload and add it to the `sp:signedPayloadData` element.

```
XmlElement signedPayloadDataElem = containerDoc.CreateElement(
    "sp", "signedPayloadData",
    "http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010");
signedPayloadElem.AppendChild(signedPayloadDataElem);
```

This results in the container document containing the following structure:

```
<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
  <sp:signatures />
  <sp:signedPayloadData />
</sp:signedPayload>
```

2.3.2 Import the payload

The payload is imported into the container document and then appended to the `sp:signedPayloadData` element. The payload can contain any XML data.

```
XmlNode payloadElem = containerDoc.ImportNode(payloadDoc.DocumentElement, true);
signedPayloadDataElem.AppendChild(payloadElem);
```

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

```
<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
  <sp:signatures />
  <sp:signedPayloadData>
    <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>
```

2.3.3 Add the ID attribute

An `id` attribute is added to the `sp:signedPayloadData` element with its value set to a GUID. A GUID is used to ensure the uniqueness of the value because the signed payload container could be composed into other XML documents that have `id` attributes. This `id` attribute is used when creating the signature references. The `sp:signedPayloadData` element, which contains the `id` attribute and the payload, will be signed as per criterion XS 3.1.1.1-4.

```
String dataGuid = Guid.NewGuid().ToString();

XmlAttribute idAttr = containerDoc.CreateAttribute("id");
idAttr.Value = dataGuid;
signedPayloadDataElem.Attributes.Append(idAttr);
```

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

```
<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
  <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 Create the signatures

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

```
foreach (X509Certificate2 signCertificate in signCertificates) {
```

The `System.Security.Cryptography.Xml.SignedXml` class is used to create signatures. This class is instantiated using the container document so the `sp:signedPayloadData` element with the `id` attribute can be resolved when computing the signature.

```
// Create the signature object
SignedXml signedXml = new SignedXml(containerDoc);
```

All signature properties have to be set first before the signature can be computed. This includes which algorithms to use, which references to sign, and which key to use.

2.3.4.1 Specify the signing private key

The private key used to create the signature must be specified. This is done by setting the `SigningKey` property to the private key. The private key must be obtained from the Windows certificate store. See appendix B.3 for details on retrieving the private key.

```
signedXml.SigningKey = signCertificate.PrivateKey;
```

2.3.4.2 Set the canonicalization algorithm

The `ds:SignedInfo` element's canonicalization method is set to exclusive canonicalization, which satisfies criterion XS 4.3.1.1-1.

```
// Specify the canonicalization method
signedXml.Signature.SignedInfo.CanonicalizationMethod =
SignedXml.XmlDsigExcC14NTransformUrl;
```

The following shows how this step is represented in an XML signature.

```
<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:SignedInfo>
  ...
</ds:Signature>
```

2.3.4.3 Set the signature algorithm

The signature algorithm is set to RSA1.5 to satisfy the criterion XS 4.3.2.1-1.

```
// Specify the signature method
signedXml.Signature.SignedInfo.SignatureMethod =
SignedXml.XmlDsigRSASHA1Url;
```

The following shows how this step is represented in an XML signature.

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    ...
    <ds:SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    ...
  </ds:SignedInfo>
  ...
</ds:Signature>
```

2.3.4.4 Add the reference

A single signing reference is added for the `sp:signedPayloadData` element which complies with XS 4.3.3.1-1. The reference is represented within the signature as the `ds:Reference` element. The canonicalization algorithm of the reference is set to exclusive canonicalization to satisfy criterion XS 4.3.1.1-1. This is the algorithm used to canonicalize the data that is being signed. The digest algorithm of the reference is set to SHA1 to satisfy criterion XS

4.3.2.1-1. This specifies the algorithm used over the canonicalized data to create the digest. The URI of the reference is set to the value of the `id` attribute that was added to the `sp:signedPayloadData` element. Since the payload is within the same document as the signature, the value of the URI is a fragment identifier. A fragment identifier starts with the `'#'` character.

```
Reference reference = new Reference();
reference.Uri = "#" + dataGuid;
reference.DigestMethod = SignedXml.XmlDsigSHA1Url;

// Add the transform
XmlDsigExcC14NTransform transform = new XmlDsigExcC14NTransform();
reference.AddTransform(transform);

// Add the reference
signedXml.AddReference(reference);
```

The following shows how this step is represented in an XML signature.

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    ...
    <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:Signature>
```

2.3.4.5 Set the `ds:KeyInfo`

The `ds:KeyInfo` is added to indicate which key was used to create the signature. A copy of the corresponding X.509 certificate of the private key that was used to create the signature is embedded within the `ds:KeyInfo`.

```
KeyInfo keyInfo = new KeyInfo();
KeyInfoX509Data kixd = new KeyInfoX509Data();
kixd.AddCertificate(signCertificate);
keyInfo.AddClause(kixd);
signedXml.KeyInfo = keyInfo;
```

The following shows how this step is represented in an XML signature.

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

2.3.4.6 Calculate the signature

The signature is created when the `ComputeSignature` method is called on the `System.Security.Cryptography.Xml.SignedXml` object. The resulting signature XML can be retrieved by calling the `GetXml` method.

```
// Calculate the signature
signedXml.ComputeSignature();

XmlElement signatureElem = signedXml.GetXml();
```

The following shows the complete structure of an XML signature.

```
<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>...</ds:SignatureValue>
<ds:KeyInfo>
  <ds:X509Data>
    <ds:X509Certificate>...</ds:X509Certificate>
  <ds:X509Data>
    <ds:KeyInfo>
</ds:Signature>

```

2.3.4.7 Add the signature to the `sp:signatures` element

The serialized signature is added to the `sp:signatures` element.

```

signaturesElem.AppendChild(signatureElem);
}

```

The signed payload container is now complete. It can be serialized or used as a DOM document.

```

<sp:signedPayload
  xmlns:sp="http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010">
  <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/J1dhDPhGwQrSb6c=</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* [ATS 5821—2010] are:

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

The inputs to the verification process are:

`XmlDocument` containerDoc

A DOM `XmlDocument` containing the signed payload container.

The output of the verification process is:

A DOM `XmlDocument` containing the payload.

Note that it is important to preserve the whitespace when reading the signed payload container into an `XmlDocument` object. If the whitespace within the payload is removed, the verification process would falsely invalidate the signature. The whitespace is preserved by setting the `PreserveWhitespace` property to `true`, as shown in the code below.

```
String containerXml = "...";
XmlDocument containerDoc = new XmlDocument();
containerDoc.PreserveWhitespace = true;
containerDoc.LoadXml(containerXml);
```

2.4.1 Get the signature elements

The code below shows how to retrieve the `ds:Signature` elements from a signed payload container using XPath [XPATH1999]. XPath is simply one way to retrieve these elements. Other ways, such as calling methods on the DOM classes, can also be used. Note that the `GetElementsByTagName` DOM method should not be used. This method returns the descendants of an element that have a particular namespace and tag name. The problem with using this method to retrieve the `ds:Signature` elements is that it will return those that are not within the `sp:signatures` element. It will cause the verification process to return false results when the payload itself contains XML signatures.

```
XmlNamespaceManager namespaceManager = new
    XmlNamespaceManager(containerDoc.NameTable);
namespaceManager.AddNamespace("ds", "http://www.w3.org/2000/09/xmldsig#");
namespaceManager.AddNamespace("sp",
    "http://ns.electronichealth.net.au/xsp/xsd/SignedPayload/2010");

XmlNodeList signatureElems =
    containerDoc.SelectNodes(
        "/sp:signedPayload/sp:signatures/ds:Signature", namespaceManager);
```

2.4.2 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 (XmlElement signatureElem in signatureElems) {
```

There are a number of aspects to checking a signature. If any of these checks fail, the signed data cannot be trusted. These checks are:

- Validating the signature;
- Verifying the credentials of the signer; and
- Checking the identity of the signer.

2.4.2.1 Validate the signature

The signature is validated using the public key of the entity that created the signature. The signer's public key is in the certificate that is included within the `ds:KeyInfo` element of the signature.

Validating the signature ensures the integrity of the data. The `CheckSignature` method of the `System.Security.Cryptography.Xml.SignedXml` class returns `true` or `false`, depending on the signature validity. 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.

```
// Load the signature
SignedXml signedXml = new SignedXml(containerDoc);
signedXml.LoadXml(signatureElem);

// Get the KeyInfo from the signature
IEnumerator keyInfoItems = signedXml.KeyInfo.GetEnumerator();
keyInfoItems.MoveNext();
KeyInfoX509Data keyInfoX509 = (KeyInfoX509Data)keyInfoItems.Current;

// Get the certificate of the signer from the KeyInfo
X509Certificate2 keyInfoCert =
    (X509Certificate2)keyInfoX509.Certificates[0];

// Check the signature is valid
bool isSignatureValid =
    signedXml.CheckSignature(keyInfoCert.PublicKey.Key);
if (!isSignatureValid) {
    throw new Exception("Signature could not be validated");
}
```

2.4.2.2 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. This will check the chain of the certificate by using the Windows certificate repository. The CA certificate must be installed in the trusted root certificate authorities.

The example code below shows how to create a trust chain and use it to verify a certificate. When the certificate cannot be verified against the trust chain, an exception is thrown. The code also shows how to disable revocation checking for development purposes. This would be enabled within a production system.

```
X509ChainPolicy chainPolicy = new X509ChainPolicy();
chainPolicy.RevocationMode = X509RevocationMode.NoCheck;

X509Chain chain = new X509Chain();
chain.ChainPolicy = chainPolicy;
bool chainResult = chain.Build(keyInfoCert);
if (!chainResult) {
    throw new Exception("Error processing trust chain: chain could not be " +
        "validated for key: " + verifyCert.Subject);
}
}
```

2.4.2.3 Check the identity

The way the identity is checked depends on the context in which the container is being used. In this example, the ID of the intended receiver is within the payload being signed. This can be checked by the receiver before the signature is validated.

```
XmlNamespaceManager payloadNamespaceManager =
    new XmlNamespaceManager(payloadDoc.NameTable);
namespaceManager.AddNamespace("ex", "http://www.example.org");

// Get the receiver ID attribute from the payload
XmlAttribute receiverIdAttr = (XmlAttribute)
    payloadDoc.SelectSingleNode(
        "/ex:pathologyReport/ex:receiver/@id", payloadNamespaceManager);

// Check if the entity that received the payload is the intended receiver
if (!receiverIdAttr.Value.Equals(receiverId)) {
    throw new Exception("Error checking the identity: IDs do not match");
}
```

2.4.3 Extract the payload

The payload can be extracted from the container using an XPath [XPATH1999] statement.

```
// Get the payload element
XmlElement payloadElem = (XmlElement)
    containerDoc.SelectSingleNode(
        "/sp:signedPayload/sp:signedPayloadData/*[1]", namespaceManager);
```

The payload can now be processed.

3 Encrypted Payload

The *XML Secured Payload Profile* [ATS 5821—2010] 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* [ATS 5821—2010].

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.

The Microsoft .NET framework has an API that implements *XML Encryption* that can be used to create and process encrypted payloads. Section 3.2 introduces this API and other libraries 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 to 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 [ATS 5821—2010].

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 [ATS 5821—2010].

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <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>zdB0/qVF1VyLTi8VN0xSqsjU/B8=</ds:X509SKI>
        </ds:X509Data>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>
uQJemj3rJTgfm4hcPe5EebpoX3O6IBv9uLuOCSP6oKXYrFnPjCjI0A7423fkRC0dqbvMkN3xOtQ94yzhF4H
py2Fs6YIEf3Xf3r6s8I+huDecpEa410ZP4/+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/9YN8wqb1IoGxiNeyGAzenxONY
oky+AVMAdqtsh6QzKx1Ze3Xnj5I8oTHMA8EfLOR/w8ObuHhQxHnfxbf6yNIZik09dS6Z41pLAVyphEotaU
v+TWE+fdhds/ti3wnxqSSA8EsDIY8d61N5P8A3ALZY73dynwWw8gJu8pWctYQYzz+ezV1ng=
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </ep:encryptedPayloadData>
</ep:encryptedPayload>
```

3.2 Libraries used

The Microsoft .NET framework contains an API for using *XML Encryption* [XENC2002]. This document explains how to encrypt and decrypt XML data using that API.

The XML Encryption API in the .NET framework is accessed by adding the `System.Security` assembly to a project. Its classes belong to the .NET namespace, `System.Security.Cryptography.Xml`. The `System.Security.Cryptography.Xml.EncryptedXml` class is the main class used to perform the encryption.

The keys used to sign data can be retrieved from the Windows certificate repository. They can be accessed using the `System.Security.Cryptography.X509Certificates.X509Store` class.

The following .NET namespaces apply to the code listings in sections 3.3 and 3.4:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Xml;
using System.Security.Cryptography.Xml;
using System.Security.Cryptography.X509Certificates;
using System.Security.Cryptography;
using System.IO;
using System.Xml.Serialization;
```

3.3 Encryption process

This section describes how to create an encrypted payload container that conforms to the *XML Secured Payload Profile* [ATS 5821–2010].

The process to create the encrypted payload container consists of:

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

The inputs to the encryption process are:

`XmlDocument payloadDoc`

A DOM `XmlDocument` containing the payload to be encrypted. This can be any XML document.

`List<X509Certificate2> encryptCertificates`

A list of one or more X.509 certificates that are to be used to encrypt the payload. These can be retrieved from the Windows certificate repository. See appendix B.3 for details on retrieving the certificates.

The output of the encryption process is:

A DOM `XmlDocument` structured according to the encrypted payload XML Schema defined in *XML Secured Payload Profile* [XSP2000].

3.3.1 Create the container document

The *XML Secured Payload Profile* [ATS 5821–2010] 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 an `XmlDocument` object.

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

3.3.1.1 Create an empty document

The container document holds the encrypted keys and the encrypted payload. This is created as an `XmlDocument`.

```
// Create the container document
XmlDocument containerDoc = new XmlDocument();
```

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:

```
// Create and add the root element
XmlElement rootElem =
  containerDoc.CreateElement("ep", "encryptedPayload",
    "http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010");
containerDoc.AppendChild(rootElem);
```

This results in the container document containing the following structure:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
</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.

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

```
// Create and add the keys element
XmlElement keysElem =
  containerDoc.CreateElement("ep", "keys",
    "http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010");
rootElem.AppendChild(keysElem);
```

This results in the container document containing the following structure:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <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:

```
// Create and add the encryptedPayloadData element
XmlElement dataElem =
  containerDoc.CreateElement("ep", "encryptedPayloadData",
```

```
"http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010");
rootElem.AppendChild(dataElem);
```

This results in the container document containing the following structure:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <ep:keys />
  <ep:encryptedPayloadData />
</ep:encryptedPayload>
```

3.3.2 Import the payload

The payload must be imported into the container document. This data will be encrypted.

```
// Import the payload into the container
XmlElement payloadElem = (XmlElement)
  containerDoc.ImportNode(payloadDoc.DocumentElement, true);
dataElem.AppendChild(payloadElem);
```

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

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <ep:keys/>
  <ep:encryptedPayloadData>
    <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>
  </ep:encryptedPayloadData>
</ep:encryptedPayload>
```

3.3.3 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 5.3.1.1-2 requires the use of AES-256 for data encryption.

The code below shows how to create a symmetric AES key with a key size of 256 bits. AES is also known as Rijndael.

```
RijndaelManaged sessionKey = new RijndaelManaged();
sessionKey.KeySize = 256;
```

3.3.4 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 reference ID is created by generating a GUID with a preceding underscore. An underscore is required because the *XML Encryption XML Schema [XEXSD2002]* specifies an `ID (NCName)` data type for the `Id` attribute. An `NCName` value must start with either a letter or an underscore.

```
string referenceId = "_" + Guid.NewGuid().ToString();
```

3.3.5 Encrypt the data

3.3.5.1 Create an EncryptedXml

The `System.Security.Cryptography.Xml.EncryptedXml` class is used to perform the encryption. This class is instantiated with the document that contains the element to encrypt.

```
EncryptedXml encryptedXml = new EncryptedXml(containerDoc);
```

3.3.5.2 Perform the encryption

The `EncryptData` method has three parameters: the element to be encrypted, the session key used to do the encryption and a content flag to specify whether to use content or element encryption. The content flag is set to false to use element level encryption, satisfying criterion XS 5.2.1.1-1. This method does not modify the element and returns the encrypted bytes.

```
byte[] encryptedBytes = encryptedXml.EncryptData(payloadElem, sessionKey, false);
```

3.3.5.3 Create the EncryptedData object

The `EncryptedData` object must be created and its properties set.

```
EncryptedData encryptedData = new EncryptedData();
encryptedData.Type = EncryptedXml.XmlEncElementUrl;
encryptedData.EncryptionMethod =
    new EncryptionMethod(EncryptedXml.XmlEncAES256Url);
encryptedData.Id = referenceId;
encryptedData.CipherData.CipherValue = encryptedBytes;
```

3.3.5.4 Add to the ep:encryptedPayloadData element

The `xenc:EncryptedData` element is added to the `ep:encryptedPayloadData` element by replacing the payload element with it in the container document.

```
EncryptedXml.ReplaceElement(payloadElem, encryptedData, false);
```

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

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <ep:keys />
  <ep:encryptedPayloadData>
    <xenc:EncryptedData Id="_ffbcc1bc-0f52-44a7-ba3b-3fb890eff3a2"
      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>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </ep:encryptedPayloadData>
</ep:encryptedPayload>
```

3.3.6 Encrypt the session key

The session key is encrypted using the public key of each receiver of the payload. All encrypted session keys are included within the encrypted payload container to enable all receivers to decrypt the data.

```
foreach (X509Certificate2 encryptCertificate in encryptCertificates) {
```

The `EncryptKey` method on the

`System.Security.Cryptography.Xml.EncryptedXml` class is used to create the encrypted key. This method takes three parameters: the session key to be encrypted, the public key used to do the encryption and a flag for using OAEP. The encryption method is set to RSA 1.5 to satisfy criterion XS 5.7.2.1-

2. The OAEP flag is set to `false` when using RSA 1.5. The encrypted key is represented using the `xenc:EncryptedKey` element.

```
byte[] encryptedKeyData = EncryptedXml.EncryptKey(sessionKey.Key,
    (RSA) encryptCertificate.PublicKey.Key, false);

// Create the encrypted key
EncryptedKey encryptedKey = new EncryptedKey();
encryptedKey.CipherData = new CipherData(encryptedKeyData);
encryptedKey.EncryptionMethod = new EncryptionMethod(EncryptedXml.XmlEncRSA15Url);
```

3.3.6.1 Add the data reference

A data reference is added to the encrypted key to link the encrypted data to the encrypted key. The `#` character at the start of the ID is to indicate that the ID is a fragment identifier. A fragment identifier is used because the encrypted key and encrypted data are within the same document.

This step satisfies criterion XS 5.7.1.1-4, which requires only one `xenc:DataReference` element whose URI attribute is a fragment identifier referencing the value of the `Id` attribute of the encrypted data.

```
DataReference dataReference = new DataReference("#" + referenceId);
encryptedKey.ReferenceList.Add(dataReference);
```

3.3.6.2 Add the ds:KeyInfo element

The `ds:KeyInfo` element must be set on the encrypted key to indicate which key was used to encrypt the encrypted key. This is used by the receiver to determine which key to use to perform the decryption. The `ds:KeyInfo` references the encryption certificate by using the subject key identifier. This satisfies criterion XS 5.7.4.1-3, which requires the subject key identifier of the encryption certificate be included in the `ds:KeyInfo` element.

```
// Set the key information for the encrypted key
KeyInfoX509Data xd = new KeyInfoX509Data();

// Get the subject key identifier from the certificate
String ski =
    ((X509SubjectKeyIdentifierExtension)encryptCertificate.Extensions["2.5.29.14"]).
        SubjectKeyIdentifier;

// Add the subject key identifier value to the KeyInfo
xd.AddSubjectKeyId(ski);
encryptedKey.KeyInfo.AddClause(xd);
```

3.3.6.3 Add the encrypted key to the ep:keys element

The created encrypted key can then be serialized, imported into the container document and added to the `ep:keys` element.

```
XmlElement encryptedKeyElem = encryptedKey.GetXml();
XmlNode importedNode = containerDoc.ImportNode(encryptedKeyElem, true);
keysElem.AppendChild(importedNode);
```

The encrypted container is now complete. It can be serialized or used as a DOM document. The following shows a complete encrypted payload container:

```
<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <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>zdB0/qVF1VyLTi8VN0xSqsjU/B8=</ds:X509SKI>
        </ds:X509Data>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
      <xenc:ReferenceList>
```

```

    <xenc:DataReference URI="#" ffbcc1bc-0f52-44a7-ba3b-3fb890eff3a2" />
  </xenc:ReferenceList>
</xenc:EncryptedKey>
</ep:keys>
<ep:encryptedPayloadData>
  <xenc:EncryptedData Id=" ffbcc1bc-0f52-44a7-ba3b-3fb890eff3a2"
    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>...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</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* [ATS 5821—2010].

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 inputs to the decryption process are:

`XmlDocument containerDoc`

A DOM `XmlDocument` containing a structure conforming to the encrypted payload XML Schema in the *XML Secured Payload Profile* [ATS 5821—2010].

`X509Certificate decryptCertificate`

The X.509 certificate with the corresponding private key that will be used to decrypt the payload. This can be retrieved from the Windows certificate repository.

The output of the decryption process is:

A DOM `XmlDocument` containing the decrypted payload.

3.4.1 Find the encrypted key

Each encrypted key contains a `ds:KeyInfo` with a subject key identifier (SKI). In this example, the encrypted key is found by comparing the SKI in the `ds:KeyInfo` with the SKI of the recipient certificate. When a match is found, the corresponding private key is used to decrypt the encrypted key.

The list of encrypted keys can be found using XPath [XPATH1999].

```

XmlNamespaceManager namespaceManager =
    new XmlNamespaceManager(encryptedPayloadDoc.NameTable);
namespaceManager.AddNamespace("ds", "http://www.w3.org/2000/09/xmldsig#");
namespaceManager.AddNamespace("xenc", "http://www.w3.org/2001/04/xmlenc#");
namespaceManager.AddNamespace("ep",
    "http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010");

XmlNodeList encryptedKeyNodes = encryptedPayloadDoc.SelectNodes(
    "/ep:encryptedPayload/ep:keys/xenc:EncryptedKey", namespaceManager);

X509SubjectKeyIdentifierExtension certificateSkiExtension =
    (X509SubjectKeyIdentifierExtension)decCert.Extensions["2.5.29.14"];

```

```

EncryptedKey encryptedKey = null;
foreach (XmlElement keyElem in encryptedKeyNodes) {
    // Load the encrypted key
    EncryptedKey currentEncryptedKey = new EncryptedKey();
    currentEncryptedKey.LoadXml(keyElem);
    IEnumerator keyInfoItems = currentEncryptedKey.KeyInfo.GetEnumerator();
    keyInfoItems.MoveNext();
    KeyInfoX509Data keyInfoX509 = (KeyInfoX509Data)keyInfoItems.Current;

    byte[] keyInfoSki = (byte[]) keyInfoX509.SubjectKeyIds[0];
    X509SubjectKeyIdentifierExtension skiExtension =
        new X509SubjectKeyIdentifierExtension(keyInfoSki, false);

    if (skiExtension.SubjectKeyIdentifier.Equals(
        certificateSkiExtension.SubjectKeyIdentifier))
    {
        encryptedKey = currentEncryptedKey;
        break;
    }
}

```

3.4.2 Decrypt the encrypted key

The encrypted key must be decrypted to get the AES session key. The encrypted key is decrypted using the private key of the recipient. The static `DecryptKey` method on the

`System.Security.Cryptography.Xml.EncryptedXml` class is used to decrypt the key. The parameters are the cipher value of the key to decrypt, the key used to perform the decryption, and an OAEP flag.

```

byte[] keyBytes = EncryptedXml.DecryptKey(encryptedKey.CipherData.CipherValue,
    (RSA) decCert.PrivateKey, false);

RijndaelManaged sessionKey = new RijndaelManaged();
sessionKey.Key = keyBytes;

```

3.4.3 Decrypt the payload

The payload is decrypted using the decrypted session key. The encrypted data XML must be loaded into an `EncryptedData` object. This involves using XPath to get the `xenc:EncryptedData` element.

```

XmlElement encryptedDataElem =
    (XmlElement)encryptedPayloadDoc.SelectSingleNode(
        "/ep:encryptedPayload/ep:encryptedPayloadData/xenc:EncryptedData",
        namespaceManager);

EncryptedXml encryptedXml = new EncryptedXml(encryptedPayloadDoc);
EncryptedData encryptedData = new EncryptedData();
encryptedData.LoadXml(encryptedDataElem);

```

The `DecryptData` method of the

`System.Security.Cryptography.Xml.EncryptedXml` class decrypts the payload using the session key. This method does not modify the document that contains the encrypted data. The decrypted data is returned in a byte array. The `ReplaceData` method is used to replace the encrypted data with the decrypted data within the container document.

```

byte[] decryptedData = encryptedXml.DecryptData(encryptedData, sessionKey);
encryptedXml.ReplaceData(encryptedDataElem, decryptedData);

```

The following shows the container document once the payload is decrypted:

```

<ep:encryptedPayload
  xmlns:ep="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010">
  <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/qVF1VyLTi8VNoxSqsjU/B8=</ds:X509SKI>
      </ds:X509Data>

```

```
</ds:KeyInfo>
  <xenc:CipherData>
    <xenc:CipherValue>XoNXHYbUogl...
  </xenc:CipherValue>
</xenc:CipherData>
<xenc:ReferenceList>
  <xenc:DataReference URI="# d450d0b6-fefc-4ac9-b73a-10f2fcbd5d63" />
</xenc:ReferenceList>
</xenc:EncryptedKey>
</ep:keys>
<ep:encryptedPayloadData>
  <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>
</ep:encryptedPayloadData>
</ep:encryptedPayload>
```

3.4.4 Extract the payload

The payload can be extracted using XPath from the container document.

```
XmlElement payloadElem = (XmlElement)
  encryptedPayloadDoc.SelectSingleNode(
    "/ep:encryptedPayload/ep:encryptedPayloadData/*[1]", namespaceManager);
```

The payload can now be processed by the application.

4 Composite Payload

4.1 Conceptual Overview

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

A common pattern for composing the containers is to sign and encrypt.

This will be described in the following section.

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; and
2. Using the signed payload as input to the encryption process, specified in section 3.3.

A Signed Encrypted Payload is processed by:

1. Decrypting the payload as per the decryption process in section 3.4 to get the signed payload; and
2. Validating and verifying the signatures as per the verification process in section 2.4.

5 Secured payloads with Web services

One use of secured payloads is with Web services. They can be used to secure the body of SOAP messages that are sent between a client and Web service. Web services are defined using WSDL and XML Schema. The WSDL specifies the operations the service supports and a binding. XML Schema is used to define the types that are used in the operations.

The secured payloads can be used within a WSDL by using the `xsd:any` element. The `xsd:any` is defined within an XML Schema what's used within the WSDL. The service artefacts are generated from the WSDL and any XML schemas used within the WSDL. The artefacts include a class that corresponds to each XML schema type. When the schema type uses an `xsd:any`, the corresponding class will contain a property that is of type `XmlElement`. This can be set to any XML fragment including the secured payload container root element.

An example XML schema complex type that uses an `xsd:any` is shown below:

```
<xsd:complexType name="SecuredReportAny">
  <xsd:sequence>
    <xsd:element name="reportId" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="receiver" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="sender" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    <xsd:any processContents="lax" minOccurs="1" maxOccurs="1"
      namespace="http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010"/>
  </xsd:sequence>
</xsd:complexType>
```

The following class is generated by the `xsd.exe` tool from the above complex type:

```
/// <remarks/>
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true,
  Namespace="http://ns.nehta.gov.au/Example/XSP/DS/Svc/SecureReceiver/1.0")]
public partial class SecuredReportAny
{
    private string reportIdField;

    ...

    private System.Xml.XmlElement anyField;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(Order=0)]
    public string reportId
    {
        get
        {
            return this.reportIdField;
        }
        set
        {
            this.reportIdField = value;
        }
    }

    ...

    /// <remarks/>
    [System.Xml.Serialization.XmlAnyElementAttribute(
  Namespace=
    "http://ns.electronichealth.net.au/xsp/xsd/EncryptedPayload/2010", Order=1)]
    public System.Xml.XmlElement Any
    {
        get
        {
```

```
return this.anyField;

}
set
{
  this.anyField = value;
}
}
}
```

Appendix A: References

- [ATS 5821—2010] Standards Australia, E-Health XML Secured Payload Profiles.
- [XDSIG2002] W3C, *XML-Signature Syntax and Processing*, W3C Recommendation, 12 February 2002, <http://www.w3.org/TR/2002/REC-xmlsig-core-20020212/>
- [XENC2002] W3C, *XML Encryption Syntax and Processing*, W3C Recommendation, 10 December 2002, <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>
- [EXXSD2002] W3C, *XML Encryption Syntax and Processing*, W3C Recommendation XML Schema, W3C, 10 December 2002. <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd>
- [XPATH1999] W3C, XML Path Language (XPath), version 1.0, W3C Recommendation, 16 November 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
- [XSXSD2002] W3C, *XML-Signature Syntax and Processing*, W3C Recommendation, 12 February 2002, XML Schema <http://www.w3.org/TR/2002/REC-xmlsig-core-20020212/xmlsig-core-schema.xsd>

Appendix B: Key management

Windows stores certificates in the Windows certificate store and provides access from applications and services through an API. The store can be managed using the Microsoft certificate MMC. This can be used to add and remove certificates. The store is split into two locations, *CurrentUser* and *LocalComputer*. Each location is then split into a number of logical stores that hold certificates with each store having a specific purpose.

B.1 Setting up the certificate MMC

The certificate MMC is used for managing certificates installed within the Microsoft certificate store.

1. Start the MMC by selecting Run from the start menu and type "mmc" in the Open: combo box and press enter, or, by entering "mmc" within a command prompt window, this opens a blank Microsoft Management Console.
2. From the "File" menu, select "Add/Remove Snap-in".
3. Press the "Add..." button and select "Certificates" from the list.
4. Select "My user account" and select "Finish", this will add a management snap-in that'll allow personal certificates to be managed.
5. Press the "Add..." button again and select "Certificates" from the list.
6. Select "Computer Account" and then select "Local computer:" on the select computer dialog.
7. Select "Close" and then "OK" on the "Add/Remove Snap-in" dialog.

B.2 Installing certificates

The following sections describe:

- Adding a certificate with a private key (PKCS12);
- Adding a certificate; and
- Adding a CA certificate.

B.2.1 Add a certificate with a private key

1. Open the certificate MMC.
2. Expand the "Certificates-Current User" tree by clicking on the plus sign.
3. Right click on the "Personal" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" and then choose the PFX file that contains the client certificate and private key and select "Next".
6. Enter the password (if any) of the private key and select "Next".
7. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Personal".

B.2.2 Add a certificate

1. Open the certificate MMC.

2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Trusted People" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the server certificate and select "Next".
6. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Other People".

B.2.3 Adding a CA certificate

1. Open the certificate MMC.
2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Trusted Root Certification Authorities" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the certificate file that is the CA and select "Next".
6. Select "Next" again and finally select "Finished", the certificate will then appear within the "Certificates" folder under "Trusted Root Certification Authorities".

B.3 Accessing the store from code

The `System.Security.Cryptography.X509Certificates.X509Store` class can be used to retrieve certificates from the Windows certificate repository from code. The store can be searched using custom search parameters. These include the name and location of the store to search, the part of the certificate to match against, and the value to search with.

The class is first instantiated with the store name and store location and then opened using the `Open` method.

```
X509Store certStore = new X509Store(StoreName.My, StoreLocation.CurrentUser);
certStore.Open(OpenFlags.ReadOnly);
```

The store is searched using the `Find` method on the `Certificates` property. The method takes the find type, find value and validity as parameters. The find type specifies the type of search, find value is the value to search on and validity determines if only valid certificates are returned by the search.

```
X509Certificate2Collection foundCerts =
    certStore.Certificates.Find(X509FindType.FindByThumbprint, findValue, valid);
certStore.Close();

X509Certificate2 matchingCert = foundCerts[0];
```

The public key of the certificate can be accessed using the `PublicKey` property, and if the certificate has a corresponding private key, it can be accessed using the `PrivateKey` property.