Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Eidgenössisches Finanzdepartement EFD

Bundesamt für Informatik und Telekommunikation BIT
Lösungszentrum
Lösungen Bern
Entwicklung 1

Matthias Rüedlinger

e-dec XML Signaturprüfung

Empfehlungen zur Umsetzung der Prüfung von digitalen Signaturen (WS-Security)

Projektname:	**e-dec**
Version:	**0.3**
Datum:	**2009-05-22**

Status	in Arbeit	in Prüfung	genehmigt zur Nutzung
	☐	☐	☑

Beteiligter Personenkreis	
Autoren:	Matthias Rüedlinger (mru)
Genehmigung:	Projektteam e-dec IDEE
Benützer/Anwender:	EZV, PL
zur Information/Kenntnis:	EZV

Änderungskontrolle, Prüfung, Genehmigung			
Wann	**Version**	**Wer**	**Beschreibung**
2009-04-24	0.1	mru	Erste Version
2009-05-14	0.2	mru	Java Code Beispiel hinzugefügt
2009-05-15	0.3	Mru, shu	Anpassungen Abschnitt CRL, Anpassung des Titels

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument richtet sich an e-dec Zollkunden und Softwarelieferanten, welche die XML Signatur einer elektronischen Veranlagungsverfügung (eVV) überprüfen wollen.

Das Dokument beschreibt, wie die Vorgaben aus der Schnittstellenbeschreibung [1] und dem Service Cotnract [2] für den EdecReceiptService umgesetzt werden können.

Die Signaturprüfung von XML Dokumenten, wurde durch die Organisation W3C spezifiziert. Die entsprechende Spezifikation **XML Signature Syntax and Processing *(*XMLDsig)** findet man unter folgender Adresse:

http://www.w3.org/TR/xmldsig-core/.

Um die die Signaturprüfung eines XML durchführen zu können, muss die entsprechende Programmiersprache oder Framework den W3C Standard **XML Signature Syntax and Processing *(*XMLDsig)** umgesetzt werden.

Die XML Signatur ist in einer SOAP Envelope eingebettet. Die Signatureinbindung im SOAP Header erfolgt nach dem WS-Security Standard.

1.1 Bemerkung

Die Aufgeführten Code Beispiele sollen lediglich als Anleitung dienen wie man bei einer eVV Antwort die Signatur und die Chain of Trust des Zertifikats überprüft. Der Code wurde nicht auf Performanz optimiert.

1.2 Referenzen

Die folgenden Dokumente enthalten Informationen über die digitale Signatur bei e-dec.

Ref	Titel	Version
[1]	Schnittstellenbeschreibung e-dec Veranlagungsverfügung (Beschreibung der eingehenden und ausgehenden Nachrichten für den Service)	1.5
[2]	Service Contract EdecReceiptService (Beschreibung der Kommunikationskanäle – Web Service und E-Mail)	1.3

2 Tools / Frameworks

Folgende Liste ist eine Auswahl an Tools / Frameworks, welche man für die XML Signaturprüfung verwenden kann:

Tool / Framework	Beschreibung	URL
Java SE 6	In Java SE 6 hat man die Möglichkeit mit den mitgelieferten API XML Signaturen zu überprüfen.	http://java.sun.com/javase/
Apache XML Security	Apache XML Security hat man eine Library mit der man XML Signaturen für Java oder C++ prüfen kann.	http://santuario.apache.org/
IAIK XML Security Toolkit (XSECT)	Ist eine kommerzielle Java Library für die XML Signaturprüfung.	http://jce.iaik.tugraz.at

3 XML Signaturprüfung mit Java

Am besten verwendet man das XML Digital Signature API (JSR 105) welche durch den Java Community Process (JCP) spezifiziert ist. Nachfolgend sind einige Implementationen aufgelistet:

- Seit Java 6 ist die XML Digital Signature API in der Java Standard Edition integriert.
- Apache XML Security ist eine freie Implementation der XML Digital Signature API.
- Das IAIK XML Security Toolkit (XSECT) ist eine kommerzielle Implementation der XML Digital Signature API.

3.1 XML Digital Signature API

Das folgende Beispiel verwendet die Standartschnittstelle der XMl Digital Signature API und somit spielt es keine grosse Rolle, für welche Implementation man sich entscheidet. Einzig bei der Registrierung des Security Providers gibt es einige Unterschiede.

Bemerkung: Bei diesem Beispiel wird die CRL der Admin PKI noch nicht überprüft.

3.1.1 Beispiel: XML Signatur prüfen

Zuerst liest man das XML Dokument ein. Wichtig ist das der DocumentBuilder **namespace aware** ist

```
InputStream is = XMLDsigClient.class.getResourceAsStream("eVVResponse.xml");

// create DocumentBuilderFactory which is Namespace aware
DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();
builderFactory.setNamespaceAware(true);

DocumentBuilder builder = builderFactory.newDocumentBuilder();
logger.info("Is DocumentBuilder NamespaceAware: " + builder.isNamespaceAware());

// parse xml file (dumped soap request)
Document xmldoc = builder.parse(is);
```

XPath muss mit einem eigenen **NamespaceContext** initialisiert werden, damit man bei einer XPath Query den Namespace verwenden kann. Der **NamespaceContextImpl** implementiert das Interface NamespaceContext und muss selber erstellt werden. (Siehe Beispiel NamespaceContext)

```
// create XPath object which has own NamespaceContext
XPath xpath = XPathFactory.newInstance().newXPath();
// with a custom NamespaceContext we can use Namespaces in our XPath query
NamespaceContext nsc = new NamespaceContextImpl();
xpath.setNamespaceContext(nsc);
```

Mit XPath kann man das **X509 Token** extrahieren. Dieses X509 Token enthält ein X509 Zertifikat das Base64 encoded ist.

```
// extract x509 Token --> xml element wsse:BinarySecurityToken
```

```
XPathExpression expr = xpath.compile("//wsse:Security/wsse:BinarySecurityToken");
Node x509Node = (Node) expr.evaluate(xmldoc, XPathConstants.NODE);
```

Wir können die Daten des X509 Token verwenden um ein X509 Zertifikat zu erstellen. Jedoch muss man beim Zertifikat noch den Anfang mit **-----BEGIN CERTIFICATE-----** und das Ende durch **-----END CERTIFICATE-----** markieren. Sonst kann man das Zertifikat nicht einlesen.

```
// X509 Token is encoded in base64
String header = "-----BEGIN CERTIFICATE-----\n";
String footer = "\n-----END CERTIFICATE-----";
// so we need a to add the certificate header and footer
// to the raw x509 data
byte[] x509data = (header + x509Node.getTextContent() + footer).getBytes();
ByteArrayInputStream bis = new ByteArrayInputStream(x509data);

// create certificate
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509Certificate cert = (X509Certificate) cf.generateCertificate(bis);
```

Hier wird das X509 Zertifikat und Chain of Trust mittels CA Zertifikat überprüft. Sollte das Zertifikat oder die Chain of Trust nicht gültig sein wird durch den **CertPathValidator.validate(…)** ein Exception geworfen.

```
// verify ca chain
// read in ca cert
is = XMLDsigClient.class.getResourceAsStream("adminca-cd-t01_BIT_CA_certificate.crt");
X509Certificate caCert = (X509Certificate) cf.generateCertificate(is);

// trusted ca cert
Set<TrustAnchor> trust = Collections.singleton(new TrustAnchor(caCert, null));
PKIXParameters params = new PKIXParameters(trust);

// Disable CRL checking since we are not supplying any CRLs
params.setRevocationEnabled(false);
// sets the time for which the validity of the certification
// path should be determined
params.setDate(new Date());

CertPath certPath = cf.generateCertPath(Collections.singletonList(cert));
CertPathValidator certPathValidator = CertPathValidator.getInstance("PKIX");
PKIXCertPathValidatorResult result = (PKIXCertPathValidatorResult) certPathValidator
    .validate(certPath, params);
```

Das XML Signaturelement wird mittels XPtah extrahiert und ein JSR 105 Provider initialisiert. Hier wird der Provider explizit initialisiert, dies ist notwendig wenn z.B. mit Apache XML Security gearbeitet wird. Mit Java 6 ist schon ein XML Digital Signature API (JSR 105) Provider registriert und dieser Schritt sollte nicht nötig sein.

```
// extract xml element ds:Signature
expr = xpath.compile("//wsse:Security/ds:Signature");
Node dsSignature = (Node) expr.evaluate(xmldoc, XPathConstants.NODE);

DOMValidateContext context = new DOMValidateContext(cert.getPublicKey(), dsSignature);

String providerName = System.getProperty("jsr105Provider",
    "org.jcp.xml.dsig.internal.dom.XMLDSigRI");

logger.info("jsr 105 provider: " + providerName);

XMLSignatureFactory factory = XMLSignatureFactory.getInstance("DOM", (Provider) Class
    .forName(providerName).newInstance());

XMLSignature signature = factory.unmarshalXMLSignature(context);
```

Die XML Signatur wird mit dem **DOMValidateContext** überprüft. Dieser besitzt den Public Key und eine Referenz auf das XML Signaturelement.

```
      // Check core validation status
      boolean coreValidity = signature.validate(context);

      if (coreValidity == false) {

        logger.error("Signature failed core validation!");
        boolean sv = signature.getSignatureValue().validate(context);
        logger.info("Signature validation status: " + sv);

        // Check the validation status of each Reference
        Iterator<Reference> i = signature.getSignedInfo().getReferences().iterator();

        for (int j = 0; i.hasNext(); j++) {
          // signature was not valid so try to find out which refrence was invalid
          Reference ref = i.next();
          boolean refValid = ref.validate(context);
          String id = ref.getURI();
          logger.info("Reference (" + j + ") with URI [" + id + "] validation status: "
               + refValid);
        }
      }
      else {
        logger.info("Signature passed core validation!");
      }
```

3.1.2 Beispiel: NamespaceContext Implementation

Dies ist die NamespaceContext Implementation welche alle nötigen Namespaces der eVV Response kennt. Der NamespaceContext wird benötigt, damit man XPath Abfragen mit den entsprechenden Präfixes machen kann. Es wird so das Mapping zwischen Präfixes und Namespaces sichergestellt.

```
public class NamespaceContextImpl implements NamespaceContext{

  public static final String NS_URI_WSSE = "http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-secext-1.0.xsd";
  public static final String PREFIX_WSSE = "wsse";
  public static final String NS_URI_SOAP_ENV = "http://schemas.xmlsoap.org/soap/envelope/";
  public static final String PREFIX_SOAP_ENV = "soap";
  public static final String NS_URI_XMLDSIG = "http://www.w3.org/2000/09/xmldsig#";
  public static final String PREFIX_XMLDSIG = "ds";
  public static final String NS_URI_EVV = "http://www.e-
dec.ch/xml/schema/edecReceiptResponse/v1";
  public static final String PREFIX_EVV = "evv";
  private Map<String, String> value = new HashMap<String, String>();

  public NamespaceContextImpl() {
    value.put(PREFIX_EVV, NS_URI_EVV);
    value.put(PREFIX_SOAP_ENV, NS_URI_SOAP_ENV);
    value.put(PREFIX_WSSE, NS_URI_WSSE);
    value.put(PREFIX_XMLDSIG, NS_URI_XMLDSIG);
  }

  public String getNamespaceURI(String prefix) {
    return value.get(prefix);
  }

  public String getPrefix(String uri) {
     throw new UnsupportedOperationException();
  }

  public Iterator<String> getPrefixes(String uri) {
    throw new UnsupportedOperationException();
  }
}
```

3.2 CRL - Certificate Revocation List;

Die CRL wird in diesem Code Beispiel noch nicht überprüft. Ein Certificate Revocation List (CRL) erhält man über die Admin PKI Homepage.

http://www.pki.admin.ch/crl.php

4 Quellen

Spezifikation XML Signature Syntax and Processing (XMLDsig)
http://www.w3.org/TR/xmldsig-core/

XMl Digital Signature API (JSR 105)
http://jcp.org/en/jsr/detail?id=105

Apache XML Security
http://santuario.apache.org/

IAIK XML Security Toolkit (XSECT)
http://jce.iaik.tugraz.at/sic/products/xml_security/xsect

Artikel: XML Signature with JSR-105 in Java SE 6
http://today.java.net/pub/a/today/2006/11/21/xml-signature-with-jsr-105.html?page=1

Artikel: Using JSR 105 with JDK 1.4 or 1.5
http://weblogs.java.net/blog/mullan/archive/2008/02/using_jsr_105_w_1.html

Präsentation: XML Security and JSR 105-106
http://www.parleys.com/display/PARLEYS/XML+Security+and+JSR+105-106