Криптографический сервис-провайдер Java LirXMLDSig Руководство программиста

ООО "ЛИССИ-Крипто"

31 августа 2011 г.

Оглавление

1 Введение

Одним из важных средств Java Platform, Standard Edition 6 (Java SE 6) является прикладной программный интерфейс поддержки ЭЦП для документов XML – The Java XML Digital Signature API. Данный API позволяет генерировать и проверять подписи XML. Подписи XML являются стандартом для ЭЦП данных в формате XML и позволяют аутентифицировать и проверять целостность данных XML в сетевых транзакциях.

К сожалению, штатные средства The Java XML Digital Signature API не поддерживают алгоритмов российской криптографии, поэтому ООО "ЛИССИ-Крипто"[1] разработало собственную реализацию криптографического сервис-провайдера LirXMLDSig, обеспечивающую такую поддержку в соответствии со спецификациями [3].

2 Обзор ЭЦП XML

RFC 2828 определяет ЭЦП как "значение, вычисленное по криптографическому алгоритму и добавленное к объекту данных таким образом, что любой получатель данных может использовать подпись для проверки источника и целостности данных".

ЭЦП XML является цифровой подписью с несколькими ключевыми свойствами. Она определяет процесс и формат для генерации ЭЦП в формате XML, а также обладает многими дополнительными возможностями. Например, она позволяет подписывать несколько частей данных – двоичных или XML – и использовать любой подходящий криптографический алгоритм подписи.

ЭЦП XML может подписывать любые данные, в формате XML или двоичные. Она может также подписывать соответствующую порцию или подмножество документа XML, а не обязательно весь документ целиком. Подписываемые данные идентифицируются с помощью Uniform Resource Identifiers (URIs). ЭЦП XML часто описываются в виде трех типов:

- Отдельная (detached) подпись является дополнительными данными, внешними по отношению к подписываемому элементу данных. Такая подпись может размещаться вне документа, например, на сетевой странице, полученной с помощью HTTP, но она может также содержаться в том же документе в качестве родственного элемента подписи.

  Пример:
  ```
  <Signature>
  ...
  </Signature>
  <DocumentData>
  ...
  </DocumentData>
  ```

- Оборачивающая (enveloping) подпись производится вокруг данных, содержащихся в самом элементе подписи.

  Пример:
  ```
  <Signature>
  <DocumentData>
  ...
  </DocumentData>
  ```

```
...
</Signature>
```

- Обернутая (enveloped) подпись – это подпись, выполненная внутри данных, содержащих сам элемент подписи, например, внутри всего документа.

  Пример:

```
<DocumentData>
...
<Signature>
...
</Signature>
</DocumentData>
```

Наверное, лучший способ описать ЭЦП XML – это пройти по шагам по примеру в деталях. Мы будем использовать обернутую подпись, генерируемую над содержимым документа XML, являющегося примером ордера заказа. Данный документ будет также использоваться в последующих разделах по использованию API. Пример XML 1 показывает содержимое ордера заказа перед подписыванием.

Пример XML 1

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder>
 <Item number="130046593231">
  <Description>Видео Игра</Description>
  <Price>300.29</Price>
 </Item>
 <Buyer id="8492340">
  <Name>Иван Петров</Name>
  <Address>
   <Street>Большая Комитетская</Street>
   <Town>Юбилейный</Town>
   <State>МО</State>
   <Country>Российская Федерация</Country>
   <PostalCode>141090</PostalCode>
  </Address>
 </Buyer>
</PurchaseOrder>
```

Результирующая обернутая попись XML, отформатированная для удобства, приведена в Примере XML 2.

Пример XML 2

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<PurchaseOrder>
 <Item number="130046593231">
  <Description>Видео Игра</Description>
  <Price>300.29</Price>
 </Item>
 <Buyer id="8492340">
  <Name>Иван Петров</Name>
  <Address>
   <Street>Большая Комитетская</Street>
   <Town>Юбилейный</Town>
   <State>MO</State>
   <Country>Российская Федерация</Country>
   <PostalCode>141090</PostalCode>
  </Address>
 </Buyer>
 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
   <CanonicalizationMethod
    Algorithm=
    "http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
   <SignatureMethod Algorithm=
    "http://www.w3.org/2001/04/xmldsig-more#gostr34102001-gostr3411"/>
   <Reference URI="">
    <Transforms>
     <Transform Algorithm=
      "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
    </Transforms>
    <DigestMethod
     Algorithm=
      "http://www.w3.org/2001/04/xmldsig-more#gostr3411"/>
    <DigestValue>
     arjK/zHBGXVV/NOrWdu6NH6QWN+bLsORFk6U4bEJQzY=
    </DigestValue>
   </Reference>
  </SignedInfo>
  <SignatureValue>
   V0EYtC0qncuM7B5DsK2A27zXN+q//ZQbwUX5j8rjjihOqLe
   ...
  </SignatureValue>
  <KeyInfo>
   <X509Data>
<X509SubjectName>
```

```
 CN=V,O=LISSI,L=Moscow,ST=MO,C=RU
</X509SubjectName>
<X509Certificate>
 MIIBgTCCAS6gAwIBAgIETOEQ2zAKBgYqhQMCAgMFADBHMQ
 ...
    </X509Certificate>
   </X509Data>
  </KeyInfo>
 </Signature>
</PurchaseOrder>
```

Заметим, что элемент подписи Signature вставлен внутрь подписываемого содержимого, что делает эту подпись обернутой. Пример XML 3 показывает элемент SignedInfo, содержащий информацию, которая действительно подписана.

Пример XML 3

```
<SignedInfo>
 <CanonicalizationMethod
  Algorithm=
  "http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
 <SignatureMethod Algorithm=
  "http://www.w3.org/2001/04/xmldsig-more#gostr34102001-gostr3411"/>
 <Reference URI="">
  <Transforms>
   <Transform Algorithm=
    "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
  </Transforms>
  <DigestMethod
   Algorithm=
    "http://www.w3.org/2001/04/xmldsig-more#gostr3411"/>
  <DigestValue>
   arjK/zHBGXVV/NOrWdu6NH6QWN+bLsORFk6U4bEJQzY=
  </DigestValue>
 </Reference>
</SignedInfo>
```

Элемент CanonicalizationMethod определяет с помощью URI алгоритм, используемый для канонизации элемента SignedInfo перед тем, как он будет подписан или проверен. Канонизация является процессом преобразования содержимого XML к однозначному представлению, называемому канонической формой, для того, чтобы ликвидировать несущественные различия в исходном формате документа, которые могут повлиять на значение ЭЦП. Канонизация необходима в силу природы XML

и способов его парсинга различными процессорами и посредниками, которые могут изменить данные без нарушения их содержательной логики, что может привести к формальному несовпадению подписи, хотя исходный и преобразованный документы логически эквивалентны. Канонизация устраняет эти синтаксические неоднозначности путем конвертирования XML к канонической форме перед генерацией или проверкой подписи.

Элемент SignatureMethod определяет с помощью URI алгоритм ЭЦП, используемый для генерации подписи, в данном случае это алгоритм ГОСТ Р34.10-2001 с ГОСТ Р34.11-94. Один или несколько элементов Reference идентифицируют подписываемые данные. Каждый элемент Reference идентифицирует данные с помощью URI. Пример XML 3 содержит единственный элемент Reference, а URI является пустой строкой `""`, которая означает корень документа – другими словами, весь документ. Ссылки Reference URI могли бы также указывать на внешние данные, например, `"http://java.sun.com"`, или на данные внутри того же самого документа, например, `"\#purchaseOrder"`. Кроме того, URI могут указывать на локальный файл по его имени, например `"PurchaseOrder.xml"`, но в этом случае в контекстах подписывания и проверки нужно указать базовый URI для указания пути к файлу, например:

```
signContext.setBaseURI("file:///G:/lirxmldsig/trunk/tests/");
```

Необязательный элемент Transforms содержит список из одного или более элементов Transform, каждый из которых описывает алгоритм, используемый для преобразования данных перед их хешированием и подписыванием или проверкой. В данном примере содержится один элемент Transform для алгоритма обернутого преобразования. Обернутое преобразование требуется для обернутых подписей, чтобы сам элемент подписи Signature не использовался при вычислении значения подписи. В противном случае, подпись включила бы саму себя в подписываемые данные, что неправильно. Другим примером является полезный алгоритм преобразования XPath Filter, позволяющий задать выражение XPath, которое выбирает подмножество подписываемых узлов.

Элемент DigestMethod определяет с помощью URI алгоритм, используемый для хеширования данных, в данном случае это ГОСТ Р34.11-94. Элемент DigestValue представлен в формате base64.

Элемент SignatureValue содержит значение подписи в формате base64 под элементом SignedInfo, как показано в Примере XML 4.

Пример XML 4

```
<SignatureValue>
 V0EYtC0qncuM7B5DsK2A27zXN+q//ZQbwUX5j8rjjihOqLe
 ...
</SignatureValue>
```

Необязательный элемент KeyInfo содержит информацию о ключе, который нужен для проверки подписи, как показано в Примере XML 5.

Пример XML 5

```
<KeyInfo>
 <X509Data>
  <X509SubjectName>
   CN=V,O=LISSI,L=Moscow,ST=MO,C=RU
  </X509SubjectName>
  <X509Certificate>
   MIIBgTCCAS6gAwIBAgIETOEQ2zAKBgYqhQMCAgMFADBHMQ
   ...
  </X509Certificate>
 </X509Data>
</KeyInfo>
```

Элемент KeyInfo может содержить различные варианты, такие как сертификаты X.509 и идентификаторы ключей Pretty Good Privacy (PGP). Обратитесь к разделу KeyInfo стандарта подписи XML для дополнительной информации о KeyInfo и типах информации, которыю он может содержать. В данном примере, KeyInfo содержит элемент X509Data, который содержит элемент X509SubjectName, идентифицирующий Distinguished Name субъекта сертификата X.509 подписанта, и элемент X509Certificate, содержащий сертификат подписанта в формате base64. Данный сертификат содержит открытый ключ, необходимый для проверки подписи. Раздел KeyInfo в The XML Signature Recommendation дает больше информации о различных типах KeyInfo.

Важно иметь в виду, что стандарт подписи XML не определяет того, как получатель устанавливает доверие к ключу, необходимому для проверки подписи. Элемент KeyInfo – это всего лишь коллекция информации, которую получатель может использовать для поиска и последующей установки доверия к данному ключу.

3 Архитектура API

The Java XML Digital Signature API был определен в рамках программы The Java Community Process, как JSR 105. API разработан для поддержки всех требуемых или рекомендуемых возможностей, определенных в The W3C Recommendation for XML-Signature Syntax and Processing. API базируется на Архитектуре Криптографических Сервис-Провайдеров Java. Это позволяет разрабатывать собственные реализации сервис-провайдеров API. Сервис-провайдеры реализуют специфический механизм XML, который идентифицирует механизм парсинга XML, используемый реализацией. Сервис-провайдер LirXMLDSig поддерживает механизм The Document Object Model (DOM).

API поддерживает шесть интерфейсных пакетов, как показано в Таблице 1.

Таблица 1. Интерфейсные пакеты в Java XML Digital Signature API

Пакет	Содержимое
javax.xml.crypto	Содержит общие классы, используемые для выполнения криптографических операций XML.
javax.xml.crypto.dom	Содержит специфические для DOM классы для пакета javax.xml.crypto.
javax.xml.crypto.dsig	Содержит классы, представляющие элементы ядра, определенные в спецификации ЭЦП XML. Особое значение имеет класс XMLSignature, позволяющий генерировать и проверять цифровую подпись XML. Класс XMLSignatureFactory является абстрактной фабрикой, используемой для создания объектов, реализующих эти интерфейсы.
javax.xml.crypto.dsig.dom	Содержит специфические для DOM классы для пакета javax.xml.crypto.dsig.
javax.xml.crypto.dsig.keyinfo	Содержит классы, представляющие структуры KeyInfo, определенные в рекомендациях по ЭЦП XML. Класс KeyInfoFactory является абстрактным классом, используемым для создания объектов, реализующих эти интерфейсы.
javax.xml.crypto.dsig.spec	Содержит классы, представляющие входные параметры для алгоритмов дайджеста, подписи, преобразования или канонизации, используемых при обработке подписей XML.

4 Подготовка к использованию ЭЦП XML

Два вопроса нужно решить, прежде чем можно будет перейти к использованию ЭЦП XML:

- Какие криптографические сервис-провайдеры будут использоваться и как их подключить к прикладной программе?
- Как будет создан и где будет храниться исходный ключевой материал – закрытый ключ и сертификат?

Поскольку в нашем случае предполагается использование российской криптографии, то для работы с ЭЦП XML мы выбираем провайдер LirXMLDSig. Для этого мы размещаем соответствующий архив LirXMLDSig.jar в папке

```
JAVA_HOME/lib/ext
```

где JAVA_HOME – это путь к установленной в системе JRE.

Ключевой материал – закрытый ключ и сертификат – мы будем хранить на токене, подготовленном для использования в качестве ключевого хранилища. Для доступа к токену мы будем использовать провайдер LirPKCS11 [2], а конфигурацию токена укажем в файле lirpkcs11.cfg. Обратитесь к документации по провайдеру LirPKCS11 для более подробной информации о его использовании.

В нашем примере мы будем использовать динамическую загрузку и конфигурирование провайдеров.

```
// Загружаем и конфигурируем провайдер LirPKCS11
// для доступа к токену.
String config = "lirpkcs11.cfg";
Provider p11 =
    new ru.lissi.security.pkcs11.LirPKCS11(config);
Security.insertProviderAt(p11, 2);

// Загружаем и конфигурируем провайдер LirXMLDSig
// для работы с ЭЦП XML.
Provider pxml =
    new ru.lissi.xml.dsig.internal.dom.XMLDSigRI();
Security.insertProviderAt(pxml, 3);
```

Можно использовать и статическое конфигурирование провайдеров путем добавления информации о них в файле

```
JAVA_HOME/lib/security/java.security
например:
...
security.provider.2=ru.lissi.security.pkcs11.LirPKCS11
C:/my_config_path/lirpkcs11.cfg
security.provider.3=ru.lissi.xml.dsig.internal.dom.XMLDSigRI
...
```

Заметим, что для того, чтобы для прикладной программы был выбран провайдер LirXMLDSig, он должен быть конфигурирован выше по приоритету, чем штатный провайдер XMLDSig.

5 Генерация обернутой ЭЦП XML

В данном разделе будет показано, как использовать API для генерации подписи XML под содержимым элемента PurchaseOrder, описанным ранее.

В данном примере будет использоваться DOM для парсинга подписываемых данных XML. Пример программы 1 показывает несколько ключевых шагов для генерации подписи XML:

Пример программы 1

```
...
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import javax.xml.crypto.dsig.*;
import javax.xml.crypto.dsig.keyinfo.*;
import javax.xml.crypto.dsig.spec.*;
import javax.xml.crypto.dsig.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.Document;
import ru.lissi.xml.dsig.internal.dom.*;
...

// Создаем DOM XMLSignatureFactory, которая будет использована
// для генерации обернутой подписи.
XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");

// Создаем Reference на обернутый документ (в данном случае,
// мы подписываем весь документ целиком, на что указавает URI ""),
// а также задаем алгоритм дайджеста GOSTR3411 и Transform ENVELOPED.
Reference ref = fac.newReference
 ("", fac.newDigestMethod(DOMDigestMethod.GOSTR3411, null),
  Collections.singletonList
   (fac.newTransform
    (Transform.ENVELOPED, (TransformParameterSpec) null)),
```

```
     null, null);

// Создаем SignedInfo.
SignedInfo si = fac.newSignedInfo
 (fac.newCanonicalizationMethod
  (CanonicalizationMethod.INCLUSIVE,
   (C14NMethodParameterSpec) null),
  fac.newSignatureMethod(
   DOMSignatureMethod.GOSTR3410_GOSTR3411, null),
  Collections.singletonList(ref));
```

Первым шагом при генерации подписи XML является конкретизация механизма XMLSignatureFactory. Метод getInstance класса XMLSignatureFactory ищет сервис-провайдер, поддерживающий DOM, и возвращает реализацию XMLSignatureFactory из провайдера с высшим приоритетом. XMLSignatureFactory – это ключевой класс в API и, как показано в Примере программы 1, используется для выработки различных компонент XMLSignature.

Второй блок Примера программы 1 создает объект Reference, который идентифицирует данные для хеширования и подписывания. Объект Reference изготавливается созданием и передачей ему в качестве параметров каждой из его компонент: URI, DigestMethod и списка преобразований Transform.

Третий блок Примера программы 1 создает объект SignedInfo, под которым генерируется подпись. Как и объект Reference, объекте SignedInfo изготавливается созданием и передачей ему в качестве параметров каждой из его компонент: CanonicalizationMethod, SignatureMethod и списка ссылок Reference.

Пример программы 2 демонстрирует шаги при построении объекта KeyInfo.

Пример программы 2

```
// Загружаем KeyStore.
// Для простоты, PIN токена указывается здесь явно.
String pin = "01234567";
KeyStore ks = KeyStore.getInstance("PKCS11", p11);
ks.load(null, pin.toCharArray());
// Получаем сертификат.
// Для простоты, используем только один сертификат в цепочке.
String alias = "MyKeypair";
Certificate[] certs = ks.getCertificateChain(alias);
X509Certificate cert = null;
if (certs[0] instanceof X509Certificate) {
 cert = (X509Certificate)certs[0];
}
// Получаем интерфейс к закрытому ключу токена
PrivateKey privKey = (PrivateKey)ks.getKey(
```

```
 alias, "01234567".toCharArray());
if (privKey == null)
{
 throw new Exception(alias +
  " could not be accessed");
}
// Создаем KeyInfo, содержащий X509Data.
KeyInfoFactory kif = fac.getKeyInfoFactory();
List x509Content = new ArrayList();
x509Content.add(cert.getSubjectX500Principal().getName());
x509Content.add(cert);
X509Data xd = kif.newX509Data(x509Content);
KeyInfo ki = kif.newKeyInfo(Collections.singletonList(xd));
```

В данном примере ключ подписи и сертификат размещены на токене, используемом в качестве KeyStore с помощью провайдера LirPKCS11. Первый блок программы извлекает сертификат подписанта X.509 из ключевого хранилища. Второй блок программы создает объект KeyInfo, используя KeyInfoFactory, которая является фабрикой для производства объектов KeyInfo. Объект KeyInfo состоит из объекта X509Data, содержащего сертификат, и из Distinguished Name субъекта этого сертификата.

Далее конкретизируется подписываемый документ, создается объект XMLSignature и генерируется подпись, как показано в Примере программы 3.

Пример программы 3

```
// Конкретизируем подписываемый документ.
DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
Document doc = dbf.newDocumentBuilder().parse
    (new FileInputStream("purchaseOrder.xml"));

// Создаем DOMSignContext и задаем PrivateKey и
// место родительского элемента для XMLSignature.
DOMSignContext dsc = new DOMSignContext
    (keyEntry.getPrivateKey(), doc.getDocumentElement());

// Создаем XMLSignature, но пока еще не подписываем.
XMLSignature signature = fac.newXMLSignature(si, ki);

// Выстраиваем, генерируем и подписываем данные обернутой подписью.
signature.sign(dsc);
```

Теперь Document содержит элемент Signature. Можно проверить его с помощью

The JAXP Transformer API для выгрузки содержимого документа в файл, как показано в Примере программы 4.

Пример программы 4

```
// Выдача результирующего документа.
OutputStream os =
new FileOutputStream("signedPurchaseOrder.xml");
TransformerFactory tf = TransformerFactory.newInstance();
Transformer trans = tf.newTransformer();
trans.transform(new DOMSource(doc), new StreamResult(os));
```

6 Генерация оборачивающей ЭЦП XML

Поскольку оборачивающая подпись включает в себя подписываемый документ, в ее структуре нужно создать соответствующую ссылку `"#PurchaseOrder"` и связать ее с документом. Мы приводим здесь текст примера целиком.

Пример программы 5

```
package ru.lissi.xml.dsig.tests;

import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import javax.xml.crypto.*;
import javax.xml.crypto.dsig.*;
import javax.xml.crypto.dsig.keyinfo.*;
import javax.xml.crypto.dsig.spec.*;
import javax.xml.crypto.dsig.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;
import ru.lissi.xml.dsig.internal.dom.*;

public class LirXMLDSig_sign_enveloping {
    public static void main(String[] args) throws Exception {
        try {
            System.out.println("LirXMLDSig Sign Enveloping Test");

            System.out.println("Creating token provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);
```

```
System.out.println("Creating XMLDSig provider LirXMLDSig");
Provider pxml =
    new ru.lissi.xml.dsig.internal.dom.XMLDSigRI();
Security.insertProviderAt(pxml, 3);

// Prepare
DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);

// Step 1
XMLSignatureFactory fac =
    XMLSignatureFactory.getInstance("DOM",
        pxml);

// Step 2
Reference ref = fac.newReference("#PurchaseOrder",
    fac.newDigestMethod(DOMDigestMethod.GOSTR3411, null));

// Step 3
Document XML = dbf.newDocumentBuilder().parse(
    new File("PurchaseOrder.xml"));
Node purchaseOrder = XML.getDocumentElement();
XMLStructure content =
    new javax.xml.crypto.dom.DOMStructure(purchaseOrder);
XMLObject obj = fac.newXMLObject(
    Collections.singletonList(content),
    "PurchaseOrder", null, null);

// Step 4
SignedInfo si = fac.newSignedInfo(
    fac.newCanonicalizationMethod(
        CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
        (C14NMethodParameterSpec) null),
        fac.newSignatureMethod(
            DOMSignatureMethod.GOSTR3410_GOSTR3411, null),
        Collections.singletonList(ref));

// Step 5
// Load the KeyStore and get the signing key
// and certificate.
System.out.println("Loading token keystore");
String pin = new String("01234567");
KeyStore ks = KeyStore.getInstance("PKCS11", p11);
```

```
ks.load(null, pin.toCharArray());

String alias = "MyKeypair";
Certificate[] certs = ks.getCertificateChain(alias);
X509Certificate x509 = null;
if (certs[0] instanceof X509Certificate) {
    x509 = (X509Certificate)certs[0];
}
if (certs[certs.length - 1]
        instanceof X509Certificate) {
    x509 = (X509Certificate)certs[certs.length - 1];
}
// getting the private key
PrivateKey privKey =
    (PrivateKey)ks.getKey(
        alias, "01234567".toCharArray());
if (privKey == null)
{
    throw new Exception(alias +
        " could not be accessed");
}

// Step 6
// Create the KeyInfo containing the X509Data.
KeyInfoFactory kif = fac.getKeyInfoFactory();
List x509Content = new ArrayList();
x509Content.add(
    x509.getSubjectX500Principal().getName());
x509Content.add(x509);
X509Data xd = kif.newX509Data(x509Content);
KeyInfo ki = kif.newKeyInfo(
    Collections.singletonList(xd));

// Step 7
XMLSignature signature = fac.newXMLSignature(si, ki,
    Collections.singletonList(obj), null, null);

// Step 8
Document doc = dbf.newDocumentBuilder().newDocument();
DOMSignContext dsc = new DOMSignContext(privKey, doc);

// Step 9
signature.sign(dsc);
```

```
            // Materialize into an xml document
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer trans = tf.newTransformer();
            trans.transform(new DOMSource(doc), new StreamResult(
                new FileOutputStream(
                    "SignedEnvelopingPurchaseOrder.xml")));
            System.out.println(
                    "LirXMLDSig Sign Enveloping Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

7 Генерация отдельной ЭЦП XML

Отдельную подпись удобно использовать, когда подписывается общедоступный документ, указываемый ссылкой HTTP. Если подписывается локальный файл, то в контекстах подписывания и проверки нужно задать значение базового URI, по которому будет найден файл. Мы приводим здесь текст примера целиком.

Пример программы 6

```
package ru.lissi.xml.dsig.tests;

import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import javax.xml.crypto.dsig.*;
import javax.xml.crypto.dsig.keyinfo.*;
import javax.xml.crypto.dsig.spec.*;
import javax.xml.crypto.dsig.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.Document;
import ru.lissi.xml.dsig.internal.dom.*;

public class LirXMLDSig_sign_detached {
    public static void main(String[] args) {

        try {
            System.out.println("LirXMLDSig Sign Detached Test");

            System.out.println("Creating token provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                    new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);
```

```
            System.out.println("Creating XMLDSig provider LirXMLDSig");
            Provider pxml =
                    new ru.lissi.xml.dsig.internal.dom.XMLDSigRI();
            Security.insertProviderAt(pxml, 3);

            XMLSignatureFactory fac =
                XMLSignatureFactory.getInstance("DOM", pxml);

            // Create a Reference to an external URI that
            // will be digested
            // using the digest algorithm
            Reference ref = fac.newReference(
            // Windows local file URI works here with
            // signContext.setBaseURI() below
            "purchaseOrder.xml",
            // HTTP reference works too
//                  "http://www.w3.org/TR/xml-stylesheet",
                fac.newDigestMethod(DOMDigestMethod.GOSTR3411, null));

            // Create the SignedInfo
            SignedInfo si = fac.newSignedInfo(
                fac.newCanonicalizationMethod
                    (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
                        (C14NMethodParameterSpec) null),
    fac.newSignatureMethod(
    DOMSignatureMethod.GOSTR3410_GOSTR3411, null),
                 Collections.singletonList(ref));

            // Load the KeyStore and get the signing key and certificate.
            System.out.println("Loading token keystore");
            String pin = new String("01234567");
            KeyStore ks = KeyStore.getInstance("PKCS11", p11);
            ks.load(null, pin.toCharArray());

            String alias = "MyKeypair";
            Certificate[] certs = ks.getCertificateChain(alias);
            X509Certificate x509 = null;


            if (certs[0] instanceof X509Certificate) {
                x509 = (X509Certificate)certs[0];
            }
            if (certs[certs.length - 1]
                    instanceof X509Certificate) {
```

```
    x509 = (X509Certificate)certs[certs.length - 1];
}

// getting the private key
PrivateKey privKey =
    (PrivateKey)ks.getKey(
        alias, "01234567".toCharArray());

if (privKey == null)
{
    throw new Exception(alias +
        " could not be accessed");
}

// Create a KeyValue containing X509 certificate
KeyInfoFactory kif = fac.getKeyInfoFactory();
List x509Content = new ArrayList();
x509Content.add(x509.getSubjectX500Principal().getName());
x509Content.add(x509);
X509Data xd = kif.newX509Data(x509Content);
KeyInfo ki = kif.newKeyInfo(Collections.singletonList(xd));

// Create the XMLSignature (but don't sign it yet)
XMLSignature signature = fac.newXMLSignature(si, ki);

// Create the Document that will hold
// the resulting XMLSignature
DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true); // must be set
Document doc = dbf.newDocumentBuilder().newDocument();

// Create a DOMSignContext and set the signing Key to the
// PrivateKey and specify where the XMLSignature
// should be inserted
// in the target document (in this case, the document root)
DOMSignContext signContext =
        new DOMSignContext(privKey, doc);
// Base URI should be set for local file
signContext.setBaseURI("file:///G:/lirxmldsig/trunk/tests/");

// Marshal, generate (and sign) the detached XMLSignature.
// The DOM Document will contain the XML Signature
// if this method returns successfully.
```

```
            signature.sign(signContext);

            // output the resulting document
            OutputStream os;
                os = new FileOutputStream(
                "SignedDetachedPurchaseOrder.xml");
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer trans = tf.newTransformer();
            trans.transform(new DOMSource(doc), new StreamResult(os));

            System.out.println(
                    "LirXMLDSig Sign Detached Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

8 Проверка ЭЦП XML

Проверка ЗЦП XML производится одинаково для всех типов подписи, поэтому мы приведем только один пример программы.

Рассмотрим использование API для проверки обернутой ЭЦП XML под содержимым только что подписанного элемента PurchaseOrder. Пример программы 7 показывает ключевые шаги проверки ЭЦП XML.

Пример программы 7

```
...
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.X509Certificate;
import javax.xml.crypto.*;
import javax.xml.crypto.dsig.*;
import javax.xml.crypto.dsig.keyinfo.*;
import javax.xml.crypto.dsig.dom.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import ru.lissi.xml.dsig.internal.dom.*;
...

// Ищем элемент Signature.
NodeList nl =
    doc.getElementsByTagNameNS(
     XMLSignature.XMLNS, "Signature");
if (nl.getLength() == 0) {
    throw new Exception("Cannot find Signature element");
}

// Создаем DOMValidateContext и задаем KeySelector
// и контекст документа.
DOMValidateContext valContext = new DOMValidateContext
    (new X509KeySelector(), nl.item(0));
// Для отдельной (detached) подписи может понадобиться
// задать базовый URI, если при подписывании использовался
// локальный файл.
```

```
//valContext.setBaseURI("file:///G:/lirxmldsig/trunk/tests/");

// Выстраиваем XMLSignature.
XMLSignature signature =
fac.unmarshalXMLSignature(valContext);

// Проверяем XMLSignature.
boolean coreValidity = signature.validate(valContext);
```

Сначала нужно найти место проверяемого элемента Signature. Одним из способов сделать это является использование метода DOM getElementsByTagNameNS, как показано в Примере программы 7. Второй блок программы создает объект DOMValidateContext, содержащий объект KeySelector и ссылку на элемент Signature. Назначение объекта KeySelector состоит в получении открытого ключа по информации из элемента KeyInfo для использования в качестве ключа проверки. В следующем разделе KeySelector рассматривается более детально. Последние две строки программы выстраивают и проверяют подпись. Метод проверки возвращает true, если подпись правильна, и false, если она не правильна.

Если подпись не правильна, необходим некоторый дополнительный код для выяснения причины неудачи, как показано в Примере программы 8.

Пример программы 8

```
// Проверяем статус основной проверки.
if (coreValidity == false) {
    System.err.println("Signature failed core validation");
    boolean sv =
     signature.getSignatureValue().validate(valContext);
    System.out.println("signature validation status: " + sv);
    if (sv == false) {
        // Проверяем статус проверки для каждой ссылки Reference.
        Iterator i =
         signature.getSignedInfo().getReferences().iterator();
        for (int j=0; i.hasNext(); j++) {
            boolean refValid =
             ((Reference) i.next()).validate(valContext);
            System.out.println(
             "ref["+j+"] validity status: " + refValid);
        }
    }
} else {
    System.out.println("Signature passed core validation");
}
```

В Примере программы 8 определяется причина неправильной подписи в виде одной из двух возможностей:

- Неправильная подпись. Криптографическая проверка подписи неудачна. Это может быть вызвано неправильным ключом проверки или изменением содержимого SignedInfo после генерации подписи.
- Неправильная ссылка или ссылки. Проверка дайджеста для ссылки неудачна. Это может быть вызвано изменением указанных ссылкой данных после генерации подписи.

Перед переходом к следующему разделу, важно заметить, что преобразования могут изменить содержимое указанных ссылкой данных перед подписанием. Следовательно, может быть важно продемонстрировать именно то содержимое, которое было подписано, проверяющему пользователю. Это можно сделать путем включения ссылочного кэширования в объекте DOMValidateContext перед проверкой подписи и вызова метода getDigestInputStream для объектов Reference, содержащихся в подписи, как показано в Примере программы 9.

Пример программы 9

```
valContext.setProperty(
"javax.xml.crypto.dsig.cacheReference", Boolean.TRUE);
// Выстраиваем XMLSignature.
XMLSignature signature = fac.unmarshalXMLSignature(valContext);
// Проверяем XMLSignature.
boolean coreValidity = signature.validate(valContext);

Iterator i = signature.getSignedInfo().getReferences().iterator();
for (int j=0; i.hasNext(); j++) {
    InputStream is = ((Reference) i.next()).getDigestInputStream();
    // Показываем данные.
}
```

KeySelector является абстрактным классом, отвечающим за поиск и выдачу ключа с использованием данных, содержащихся в объекте KeyInfo. В Примере программы 7 был передан объект X509KeySelector, являющийся очень простой реализацией KeySelector, которая ищет и возвращает открытый ключ сертификата X.509, как показывает Пример программы 10.

Пример программы 10

```
public static class X509KeySelector extends KeySelector {
    public KeySelectorResult select(KeyInfo keyInfo,
                                    KeySelector.Purpose purpose,
```

```
                                    AlgorithmMethod method,
                                    XMLCryptoContext context)
        throws KeySelectorException {
        Iterator ki = keyInfo.getContent().iterator();
        while (ki.hasNext()) {
            XMLStructure info = (XMLStructure) ki.next();
            if (!(info instanceof X509Data))
                continue;
            X509Data x509Data = (X509Data) info;
            Iterator xi = x509Data.getContent().iterator();
            while (xi.hasNext()) {
                Object o = xi.next();
                if (!(o instanceof X509Certificate))
                    continue;
                final PublicKey key =
                 ((X509Certificate)o).getPublicKey();
                // Make sure the algorithm is compatible
                // with the method.
                if (algEquals(method.getAlgorithm(),
                 key.getAlgorithm())) {
                    return new KeySelectorResult() {
                        public Key getKey() { return key; }
                    };
                }
            }
        }
        throw new KeySelectorException("No key found!");
    }

    static boolean algEquals(String algURI, String algName) {
        if ((algName.equalsIgnoreCase("DSA") &&
            algURI.equalsIgnoreCase(SignatureMethod.DSA_SHA1)) ||
            (algName.equalsIgnoreCase("RSA") &&
            algURI.equalsIgnoreCase(SignatureMethod.RSA_SHA1)) ||
            (algName.equalsIgnoreCase("1.2.643.2.2.19") &&
            algURI.equalsIgnoreCase(
             DOMSignatureMethod.GOSTR3410_GOSTR3411))
        {
            return true;
        } else {
            return false;
        }
    }
}
```

Это очень простая реализация KeySelector, которая возвращает открытый ключ из первого сертификата X.509 и ищет в X509Data. Она предназначена только для демонстрационных целей и не предполагает использования в реальных приложениях. Более полные реализации селекторов ключа X.509 должны проверять другие типы X509Data и устанавливать доверие к ключу проверки с помощью ключевого хранилища доверенных ключей или путем поиска и проверки цепочки сертификатов от доверенного источника к сертификату, содержащему открытый ключ. Обратитесь к The Java PKI Programmer's Guide для дополнительной информации о доверенных источниках и к Java API, которые могут использоваться для установки доверия к ключам.

9 Сообщения и отладка

Реализация ЭЦП XML Java SE 6 включает расширенную поддержку сообщений, которая, будучи включенной, предоставляет дополнительную отладочную информацию для обработки ошибок при генерации и проверке подписей. Отладочные сообщения используют средство логгирования JDK java.util.logging.

Для включения отладочных сообщений ЭЦП XML, нужно сначала конфигурировать средство логгирования так, чтобы выдавались сообщения ЭЦП XML. Это можно сделать непосредственным редактированием умалчиваемого файла JRE logging.properties или созданием своего собственного файла и установкой ссылки на него в свойстве java.util.logging.config.file, например:

```
java -Djava.util.logging.config.file=logging.properties ...
```

где logging.properties содержит следующий код:

```
handlers = java.util.logging.ConsoleHandler
.level= INFO
java.util.logging.ConsoleHandler.level = FINER
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter
ru.lissi.dsig.internal.level = FINER
com.sun.org.apache.xml.internal.security.level = FINER
```

Этот код включает отладочные сообщения уровня FINER и выше на консоль. Все другие компоненты будут выдавать отладочные сообщения уровня INFO и выше.

Данный документ не описывает детально каждое сообщение, но в Таблице 2 приводятся некоторые наиболее полезные сообщения.

Таблица 2: Некоторые полезные отладочные сообщения

Отладочное сообщение	Объяснение
`[java] FINER: Pre-digested input: ...`	Данное сообщение показывает содержимое данных перед их хешированием. Это полезно для отладки ошибок при проверке ссылок на данные.
`[java] FINE: Expected digest: ...` `[java] FINE: Actual digest: ...`	Данное сообщение показывает ожидаемое и действительное значения дайджеста для элемента Reference в формате base64. Это также полезно для отладки ошибок при проверке ссылок на данные.
`[java] FINE: Canonicalized SignedInfo: ...`	Данное сообщение показывает канонизированный элемент SignedInfo перед подписанием. Это полезно для отладки ошибок канонизации и проверки подписи.

10 Заключение

Целью данного документа было введение в использование API и демонстрация основных шагов при генерации и проверке ЭЦП XML с использованием криптографического сервис-провайдера LirXMLDSig.

11 Ссылки

1. Официальный сайт ООО "ЛИССИ-Крипто". – http://www.lissi-crypto.ru/.

2. Криптографический сервис-провайдер Java LirPKCS11. Руководство программиста. – ООО "ЛИССИ-Крипто 2011.

3. S. Leontiev, P. Smirnov, A. Chelpanov. "Using GOST 28147-89, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms for XML Security – CRYPTO-PRO, May 31, 2010. – http://tools.ietf.org/html/draft-chudov-cryptopro-cpxmldsig-07.