Manual

High-Level API Version 1.1

30 July 2010, Graz, Austria

Contents

Examples

Tables

1 Introduction

This software library offers a simple API for key management, digital signatures and encryption. It is written solely in the Java Programming Language and developers can use it by integrating its JAR files. Beside ease of use, this product offers high data throughput. It is especially suitable for server applications, but it fits various other applications as well.

After the introduction, there is an overview of the system requirements, the features, the usage of the library and example source code. The following chapters each describe one family of features. The second chapter is about key management features, the third about signatures and encryption and the fourth about certificate validation.

2 Overview

This section gives an overview over the system requirements, the features, the installation and basic usage of this library with example code.

2.1 System Requirements

These are the minimum requirements for working with this library:

- A Java Runtime Environment that is compatible with Java 1.4, Java 5 or Java 6
- IAIK-JCE version 3.16 or later (included)
- IAIK-CMS version 3.3 or later (included) for CMS and S/MIME signature creation and verification
- IAIK XML Security Toolkit (XSECT) version 1.10 or later (included) for XML Signature verification

2.2 Features

The most important features of this library are

- Exceptionally simple API
  - Supporting the most common use-cases
  - Easy integration with more comprehensive libraries
- High-performance
  - Processing of data on-the-fly (stream processing)
  - Minimum memory footprint
  - High throughput
- Key management
  - Import of private key files in PKCS#12, PFX and Java key store format
  - Generation of key-pairs, self-signed certificates and certificate signing requests
  - Export of private keys in PKCS#12 and PFX format
- Digital signatures
  - Create and verify CMS ([CMS]) and PKCS#7 signatures
  - Create and verify S/MIME ([SMIME]) signatures, interoperable with Microsoft Outlook, Microsoft Outlook Express and Mozilla Thunderbird
  - Create and verify detached, enveloping and enveloped XML signatures ([XMLDSIG]) with signed properties according to XAdES ([XADES])
- Encryption and decryption
  - Encrypt and decrypt CMS and PKCS#7 enveloped data

  - Encrypt and decrypt S/MIME encrypted data and e-mails, interoperable with Microsoft Outlook, Microsoft Outlook Express and Mozilla Thunderbird

- Certificate chain building and validation
  - X.509 certificates according to PKIX ([PKIX]) using OCSP (requires Java 5 or newer) and CRLs

2.3 Architecture

The API consists of a single package with the name `iaik.hlapi`. The most important classes in this package are:

- `KeyAndCertificate`
  supports key management, e.g. read and write PKCS#12 and PFX files, load and store Java key stores, generate key-pairs and certificate signing requests.
- `SignerEncrypter`
  performs signature creation and encryption. With the concrete implementation classes
  - `CMSSignerEncrypter`
    for signing and encrypting data in the CMS and PKCS#7 format.
  - `SMimeSignerEncrypter`
    for signing and encrypting data in the S/MIME format.
  - `XMLSignerEncrypter`
    for signing data with the XML Signature format.
- `DecrypterVerifier`
  performs decryption and signature verification. With the concrete implementation classes
  - `CMSDecrypterVerifier`
    for decrypting data and verifying signatures in the CMS and PKCS#7 format.
  - `SMimeDecrypterVerifier`
    for decrypting data and verifying signatures in the S/MIME format.
  - `XMLDecrypterVerifier`
    for verifying signatures in the XML Signatures format.
- `CertValidator`
  builds and validates certificate chains. With the concrete implementation class
  - `PkixCertValidator`
    for certificate validation according to PKIX.

2.4 Installation

To use this library, a developer has to include these files in the class path

- `iaik_hlapi.jar`
- signed version of `iaik_jce.jar` (or `iaik_jce_full.jar` alternatively)
- `iaik_cms.jar` for CMS and S/MIME support
- `iaik_xsect.jar` for XML signature verification

For XML signature verification, include these Apache XML libraries[1] in addition

- `xalan.jar`
- `serializer.jar`
- `xml-apis.jar` (only required for Java 1.4)
- `xercesImpl.jar` (only required for Java 1.4)

For Java 1.4, it may be necessary to place the Apache XML libraries in the `endorsed` or extensions (`ext`) directory of the Java runtime.

If an application uses only a subset of the functionality, it may exclude unused libraries. For example, if an application does not use `XMLDecrypterVerifier`, it may omit `iaik_xsect.jar`, `xml-apis.jar`, `xalan.jar`, `xercesImpl.jar` and `serializer.jar`.

Please consider installing the *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files* in the Java Runtime Environment to enable support for all algorithms and key strengths. Usually, the manufacturer provides these policy files for download. Installation typically requires replacing the two JAR files `local_policy.jar` and `US_export_policy.jar` in the `lib/security` directory of the Java Runtime. Please refer to the JCE documentation of your Java Runtime for further details.

Optionally, the application may include the following IAIK crypto libraries[2]:

- `iaik_ecc.jar` to enable elliptic curve cryptography
- `iaikPkcs11Wrapper.jar` and the signed version of `iaikPkcs11Provider.jar` for crypto hardware support (smart cards or HSMs)

The application may use additional JAR files as needed.

2.5 Provider Installation

Per default, the High-Level API installs the IAIK-JCE provider automatically in the JCA framework. It also installs the IAIK XSECT provider automatically if the application verifies an XML signature. Usually, the application does not need to care about this.

The High-Level API installs the IAIK-JCE provider in the last position, i.e. using `Security.addProvider(new IAIK())`. If an application uses long keys in Java

[1] The Apache license for these Apache XML libraries is included in the same directory as the JAR files.
[2] Using these additional libraries requires additional licenses. Please contact http://jce.iaik.tugraz.at or jce-sales@iaik.tugraz.at.

1.4, e.g. 4096 bit RSA keys, the SUN JCA providers installed at a lower index (and thus having higher priority) may not support these long keys. Installing unlimited strength jurisdiction policy files won't help either because these restrictions are in the SUN implementation itself. In this case, the application can install the IAIK-JCE provider itself at a lower index **before** using the High-Level API the first time. It may use the method `IAIK.addAsProvider()` for this purpose. Then, the Java Runtime will perform the RSA operations using the IAIK-JCE provider, which does not have a key-size restriction.

2.6 Detailed API Documentation

This document provides installation and basic usage instructions. A developer can refer to the JavaDoc documentation to see detailed information about classes, interfaces, methods and parameters.

2.7 Source Code Examples

In the `demo` directory, you can find examples. The source code is in the directory tree `demo/src`. Ensure that the `demo` directory is the working directory when running these examples. Otherwise, the examples will not find the supplementary data (e.g. keys and certificates) in the `demo/data` directory. Please note that some of these examples use additional IAIK libraries like the IAIK Elliptic Curve Provider (IAIK-ECC) and the IAIK PKCS#11 Provider. All such additional libraries in the demos are for evaluation only and must not be used for any other purpose, using any of these libraries requires additional licenses. Refer to http://jce.iaik.tugraz.at or jce-sales@iaik.tugraz.at for further information.

3 Key Management

The class `KeyAndCertificate` provides features for key management. All text within this section refers to this class unless otherwise stated.

3.1 Reading Private Keys from Files

Security software often stores private keys with certificate chains in files using the PKCS#12 (*.p12) or PFX (*.pfx) format. For example, Microsoft Windows, Microsoft Internet Explorer, Netscape, Mozilla Thunderbird or Mozilla Firefox can read and write this file format. An application can read such private key files using the method `readPkcs12`.

```
String fileName = "private-key.pfx";
char[] password = "fx56g35h".toCharArray();
KeyAndCertificate keyAndCert = KeyAndCertificate.readPkcs12(
    new FileInputStream(fileName), password);
```

Example 3-1: Reading a private key from a PKCS#12 or PFX file

To read keys from any type of Java `KeyStore`, the application may use `readKeyStore`.

```
String fileName = "keystore.jks";
char[] password = "changeit".toCharArray();
KeyAndCertificate keyAndCert = KeyAndCertificate.readKeyStore(
   new FileInputStream(fileName), "JKS", null, password);
```

Example 3-2: Read keys from a Java KeyStore

The key store type is `JKS` in this example, which is the default file key store format used by SUN implementations. However, any other type is also accepted, e.g. `IAIKKeyStore` or even `PKCS11` if you have installed the IAIK PKCS#11 Provider for smart card and HSM support (have a look at the `SmartCardSigningDemo`).

3.2 Generate a Key-Pair with a Self-Signed Certificate

If an application needs a new key-pair, it can generate a new one together with a self-signed certificate. It may set a dummy value for the certificate subject, if it does not need a certificate, however, the method `generateSelfSigned` will always create a certificate. This certificate is useful for keeping the public key. This way, the application can store the key together with the self-signed certificate while waiting for a CA to provide a final certificate.

```
String subject =
  "CN=Karl Scheibelhofer,O=IAIK,C=AT,EMAIL=kscheibelhofer@iaik.at";
KeyAndCertificate keyAndCert =
  KeyAndCertificate.generateSelfSigned("RSA", null, 1024, subject);
```

Example 3-3: Generating a key pair and a self-signed certificate

3.3 Generate a Certificate Signing Request

To get a certificate from a CA, a client has to send a certificate-signing request (CSR). An application can generate such a request in PKCS#10 format by using the method `generateCertificateRequest`. Note that this method uses the subject of its current certificate as the subject for the request. Thus, the application can generate a self-signed certificate in advance. While waiting for the certificate from the CA, the application can store the private key with the self-signed certificate and replace it later.

In addition, the application may text-encode the request in PEM format using the `pemEncode` method.

```
String subject =
  "CN=Karl Scheibelhofer,O=IAIK,C=AT,EMAIL=kscheibelhofer@iaik.at";
KeyAndCertificate keyAndCert =
  KeyAndCertificate.generateSelfSigned("RSA", null, 1024, subject);
byte[] encodeRequest = keyAndCert.generateCertificateRequest();
String textRequest = KeyAndCertificate.pemEncode(encodeRequest,
  "-----BEGIN NEW CERTIFICATE REQUEST-----",
  "-----END NEW CERTIFICATE REQUEST-----");
```

Example 3-4: Generating a certificate-signing request

3.4 Storing Key and Certificate in a File

An application can store a private key together with a certificate chain in a PKCS#12 file with the `storePkcs12` method. It does not matter if there is only a single certificate (e.g. a self-signed) or a complete certificate chain in the `KeyAndCertificate` object.

```
String subject =
  "CN=Karl Scheibelhofer,O=IAIK,C=AT,EMAIL=kscheibelhofer@iaik.at";
KeyAndCertificate keyAndCert =
  KeyAndCertificate.generateSelfSigned("RSA", null, 1024, subject);
OutputStream fileStream = new FileOutputStream("private-key.pfx");
keyAndCert.storePkcs12(fileStream, "4q71fm9x".toCharArray());
```

Example 3-5: Storing a private key and certificate chain to a PKCS#12 or PFX file

The method `writeKeyStore` allows storing keys to a file with a different format than PKCS12. Similar to `readKeyStore`, it supports all supported types of Java key stores.

```
KeyAndCertificate key = ...
OutputStream fileStream = new FileOutputStream("keystore.jks");
KeyAndCertificate.writeKeyStore(new KeyAndCertificate[] {key},
  fileStream, "JKS", null, "1234".toCharArray());
```

Example 3-6: Store keys in an arbitrary Java key store

4 Digital Signatures and Encryption

`SignerEncrypter` is the abstract base class for all implementations that support signing and encryption of data. A concrete implementation uses a certain format for encoding the signed and/or encrypted data, e.g. signed e-mails according to S/MIME.

```
SignerEncrypter signerEncrypter = ...
KeyAndCertificate signingKey = ...
signerEncrypter.setSigningKey(signingKey);
X509Certificate recipientCert = ...
signerEncrypter.addRecipient(recipientCert);
byte[] data = ...
byte[] encSigData = signerEncrypter.process(data);
signerEncrypter.dropSigningKey();
```

Example 4-1: Signing and encryption of data

To sign and encrypt data, an application instantiates a suitable `SignerEncrypter` object, sets the signing key, specifies the certificate of the recipient, and processes the data. The application can provide multiple recipient certificates, which results in encrypted data that each recipient can decrypt. If it does not specify a recipient certificate, the `SignerEncrypter` will not encrypt the data, i.e. it will only sign the data, if a signing key was set. Similarly, the object will not sign the data, if the application did not set a signing key. If the application sets neither a signing key nor a recipient certificate, the `SignerEncrypter` object will pass through the data without signing or encryption. However, the data will usually be formatted and enveloped depending on the concrete implementation.

After processing a block of data, the application can reuse the `SignerEncrypter` to process another block. Optionally, the application can tell the object to drop all references to the signing key when it has finished using it.

```
DecrypterVerifier decrypterVerifier = ...
KeyAndCertificate decryptionKey = ...
decrypterVerifier.registerDecryptionKey(decryptionKey);
byte[] encSigData = ...
byte[] data = decrypterVerifier.process(encSigData);
CertValidator validator = ...
X509Certificate[] signerCertChain =
    decrypterVerifier.verify(validator);
Date signingTime = decrypterVerifier.getSigningTime();
```

Example 4-2: Decryption and verification of signatures

`DecrypterVerifier` is the abstract class that does the reverse operation, i.e. decrypting data and verifying digital signatures. For decryption, first, the application registers its decryption key. It may also register multiple keys; the decryption object will select the right one automatically. If the application handles only non-encrypted data, it can omit this step. Then, the application processes the signing and/or encrypted data and receives the decrypted data as result. During decryption, the `DecrypterVerifier` also verifies the digital signature of the data. If available, the object returns the certificate chain of the signer and the signing time (see chapter 5 for information about certificate validation). The application can reuse the object to process more encrypted and signed data. Finally, the application can tell the object to drop all references to the decryption keys.

The application can provide a `CertValidator` to the `verify` method. The `DecrypterVerifier` will use it to validate the signing certificate with respect to the signing time. If there is no signing time available, it will use the current time.

`SignerEncrypter` and `DecrypterVerifier`, both support processing of single data blocks given as byte arrays.

```
SignerEncrypter signerEncrypter = ...
byte[] data = ...
byte[] processedData = signerEncrypter.process(data);
```

Example 4-3: Processing data as a single block

Both classes also have a second processing method that handles streams. The `SignerEncrypter` takes an `OutputStream` and returns another one. The application writes the content data to this stream and the `SignerEncrypter` processes the data directly, i.e. it writes the signed and/or encrypted data to the argument stream. When the application closes the stream, the object writes the remaining data to the underlying stream and closes it. The `SignerEncrypter` only buffers a small amount of data during operation, resulting in little memory overhead and fast processing.

```
SignerEncrypter signerEncrypter = ...
OutputStream resultOut = ...
OutputStream dataStream = signerEncrypter.process(resultOut);
while (...) {
  byte[] dataPiece = ...
  dataStream.write(dataPiece);
}
dataStream.close();
```

Example 4-4: Processing data using streams

The `DecrypterVerifier` offers a similar processing method using `InputStream` objects. The application can read the decrypted and verified data from the returned stream and then get information about the verified signature, if there was one.

It is better to use stream processing for processing larger documents.

4.1 CMS Signatures and Encryption

The `CMSSignerEncrypter` class implements `SignerEncrypter`. It creates CMS `SignedData` objects for signed data and `EnvelopedData` object for encryption. Thereby, the data signed first and then encrypted. If the application does not set either a signing key or a recipient certificate, the `CMSSignerEncrypter` will wrap the processed data in a CMS `Data` object. When signing, the `CMSSignerEncrypter` adds the signed attributes content type, signing time, message digest, and signing certificate (as described in [ESSCERT]) to the signature automatically. The signatures are CAdES Basic Electronic Signature (CAdES-BES) as specified in [CADES].

The `CMSDecrypterVerifier` does the reverse operation of the `CMSSignerEncrypter`. It decrypts the data and verifies the digital signature if there is one. It can also process a CMS Data object, which is not encrypted and unsigned. The verification also checks signed attributes for consistency.

4.2 S/MIME Signatures and Encryption

A `SMimeSignerEncrypter` signs and encrypts data using the S/MIME ([SMIME]) format, i.e. it can create signed and/or encrypted e-mail messages. In addition to the methods of `SignerEncrypter`, the `SMimeSignerEncrypter` offers additional methods. There are methods to set the sender e-mail address, recipient addresses, the subject field, other header fields, the content type (e.g. `text/xml`) and the character set of the content data. Setting the signer key automatically extracts the e-mail address from the signing certificate and adds a `FROM` header with this value. When adding recipient certificates, similarly, this class adds the e-mail address to the header field `TO`. The `SMimeSignerEncrypter` adds the same signed attributes to a signature as the `CMSSignerEncrypter`.

The class `SMimeDecrypterVerifier` decrypts messages and verifies signatures according to S/MIME. It is the counterpart to `SMimeSignerEncrypter`. In addition to the methods of `DecrypterVerifier`, it provides methods to get information about the e-mail message like sender and subject.

4.3 XML Signatures

`XMLSignerEncrypter` creates XML signatures. The application can specify the URI that refers to the content it wants to sign. Moreover, it can set the content type (e.g. `text/xml`) of the signed content. By default, the class automatically includes signed signature properties according to the ETSI XAdES standard version 1.1.1, 1.2.2 or 1.3.2 (default), which can be selected. The supported properties are signing time, signing certificate, signature policy implied and the data object format, which contains the provided content type (MIME type). The default signature form is an enveloping signature that contains the signed content inside a `dsig:Object` element.

```
XMLSignerEncrypter signer = new XMLSignerEncrypter();
signer.setSigningKey(signingKey);
signer.setReferenceURI(
  "http://www.iaik.tugraz.at/research/sha2_testvectors.zip");
signer.setContentType("application/x-zip-compressed");
byte[] data = ... // the content of sha2_testvectors.zip
byte[] encodedXmlSignature = signer.process(data);
```

Example 4-5: Creation of an enveloping XML signature

To create a detached signature, the application tells the signer not to include the data using `setIncludeData(false)`. To exclude signed properties, call `setIncludeSignedProperties(false)`.

```
XMLSignerEncrypter signer = new XMLSignerEncrypter();
signer.setSigningKey(signingKey);
signer.setReferenceURI(
  "http://www.iaik.tugraz.at/research/sha2_testvectors.zip");
signer.setContentType("application/x-zip-compressed");
signer.setIncludeData(false);
signer.setIncludeSignedProperties(false);
byte[] data = ... // the content of sha2_testvectors.zip
byte[] encodedXmlSignature = signer.process(data);
```

Example 4-6: Creation of a detached XML signature

In addition, an application can set a list of encoded `Transform` elements that will be included in the reference that refers to the signed data. The application must provide these `Transform` elements in canonical form. In general, the application must provide the data in the form that is input to the hash computation, i.e. after all transformations. The `XMLSignerEncrypter` does not perform any transformations internally, even if the application specifies `Transform` elements.

```
XMLSignerEncrypter signer = new XMLSignerEncrypter();
signer.setSigningKey(signingKey);
signer.setReferenceURI("http://www.iaik.tugraz.at/test/testdoc.xml");
signer.setContentType("text/xml");
signer.setReferenceTransformElements("<dsig:Transform " +
  "Algorithm=\"http://www.w3.org/TR/2001/REC-xml-c14n-20010315\">" +
  "</dsig:Transform>");
signer.setIncludeData(false);
byte[] data = ... // the canonicalized(!) content of testdoc.xml
byte[] encodedXmlSignature = signer.process(data);
```

Example 4-7: Including a Transformation element in an XML Signature

If an application wants to include the resulting XML signature in another XML document that uses namespaces, the `XMLSignerEncrypter` must include these namespaces when calculating the signature value. The application can specify additional namespaces that are in the context of the resulting `dsig:Signature` element. This way, embedding the XML signature into a document does not break the signature, i.e. creating an enveloped XML signature.

The application can also create XML Signatures that use Exclusive Canonical XML instead of normal Canonical XML in the element `CanonicalizationMethod` and as a transform in the reference to the signed properties. This feature is useful in environments that prefer Exclusive Canonical XML, like in a signed SAML assertion.

```
XMLSignerEncrypter signer = new XMLSignerEncrypter();
signer.setSigningKey(signingKey);
signer.setReferenceURI("");
signer.setContentType("text/xml");
signer.setReferenceTransforms("<dsig:Transforms><dsig:Transform " +
  "Algorithm=\"http://www.w3.org/TR/1999/REC-xpath-19991116\">" +
  "<dsig:XPath>ancestor-or-self::tns:Data</dsig:XPath>" +
  "</dsig:Transform><dsig:Transform " +
  "Algorithm=\"http://www.w3.org/TR/2001/REC-xml-c14n-20010315\">" +
  "</dsig:Transform></dsig:Transforms>");
final OutputStream out = ...
out.write(
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n".getBytes("ASCII"));
out.write(("<tns:Document xmlns:tns=" +
  "\"http://iaik.tugraz.at/namespace/java/hlapi-1234567890#\">" +
  "<tns:Data xmlns:tns=" +
  "\"http://iaik.tugraz.at/namespace/java/hlapi-1234567890#\">" +
  "<tns:SimpleElement>Test String</tns:SimpleElement>" +
  "</tns:Data>").getBytes("ASCII"));
OutputStream signerStream = signer.process(
  new FilterOutputStream(out) {
    public void close() { }
  });
byte[] data = ... // the element tns:Data
signerStream.write(data);
signerStream.close();
out.write("</tns:Document>".getBytes("ASCII"));
```

Example 4-8: Creation of an enveloped XML signature

XML Signature verification is simpler than the creation. An `XMLDecrypterVerifier` verifies the first XML signature contained in the given XML document, which must be valid. The verifier contains the required XML Signature and XAdES schema files for validation. After verification, the application can get the signing time and the content type of the signed data, if the signature contained this information.

```
XMLDecrypterVerifier verifier = new XMLDecrypterVerifier();
Byte[] signature = ... // XML document containing the XML Signature
byte[] data = verifier.process(signature);
CertValidator validator = ...
X509Certificate[] signerCertChain = verifier.verify(validator);
Date signingTime = verifier.getSigningTime();
String contentType = verifier.getContentType();
```

Example 4-9: Verify an XML signature

In addition, the application may specify additional schemas via URLs. This is especially useful if the provided XML document requires an additional schema for validation.

```
XMLDecrypterVerifier verifier = new XMLDecrypterVerifier();
verifier.addSchemaURL("data/signed-data2.xsd");
verifier.addSchemaURL("appl-schema.xsd");
```

Example 4-10: Adding application-specific schemas

The current implementation of this library does not support XML encryption and decryption.

4.4 Algorithm Selection

All classes for signing and encryption select suitable algorithms automatically based on the given signing and encryption keys. The selection is based on the algorithm and key size recommendations given in [SP 800-57] (RSA, DSA) and [SP 800-57, ECC-CMS] (ECDSA).

Table 4-1 shows the hash algorithm selected for signature operations depending on the public-key size and public-key algorithm. For RSA the key size l corresponds to the length (in bits) of the modulus n, for DSA the key size l corresponds to the size (in bits) of the prime modulus p, and for ECDSA the key size l corresponds to the size (in bits) of the order of the base point G.

RSA, DSA[*]		ECDSA	
l ≤ 1024	SHA-1	l ≤ 223	SHA-1
1024 < l ≤ 3072	SHA-256	224 ≤ l ≤ 383	SHA-256
3072 < l ≤ 7680	SHA-384	384 ≤ l ≤ 511	SHA-384
l > 7680	SHA-512	l > 511	SHA-512

[*] FIPS 186-3 defines DSA only until prime modulus length l = 3072

Table 4-1: Selected hash algorithms for signing

Table 4-2 shows the selected symmetric content encryption algorithm based on the public-key size, or that of the smallest key if there is more than one recipient. The mode of operation is always cipher-block-chaining (CBC).

Key Size	Content Encryption Algorithm
l ≤ 2048	3DES CBC
2048 < l ≤ 3072	AES-128 CBC
l > 3072	AES-256 CBC

Table 4-2: Selected symmetric cipher for encryption with RSA

5 Certificate Validation

The class `PkixCertValidator` implements the base class `CertValidator` for certificate validation. It validates certificates according to the PKIX standard RFC 3280 using revocation checking with OCSP and CRLs. OCSP support requires a Java 5 or newer runtime. By default, any policy is accepted, policy mapping is not inhibited, no explicit policy is required and the any policy OID is not inhibited. The current implementation does not allow setting parameters for policy checking. Moreover, the current implementation does not support automatic retrieval of additional certificates from external stores like LDAP server during certificate chain construction.

First, the application sets one or more trusted root certificates. In addition, it can add additional certificates that may be required for building a certificate chain, but the `CertValidator` does not consider these certificates explicitly trusted. They inherit their trust status from their issuer certificate. Finally, the application validates a certificate or certificate chain. As a result, it gets the certificate that served as the trusted root during validation.

```
X509Certificate trustedRootCertificate1 = ...
X509Certificate trustedRootCertificate2 = ...
X509Certificate[] certificateChain = ...
CertValidator validator = new PkixCertValidator();
validator.addTrustedCertificate(trustedRootCertificate1);
validator.addTrustedCertificate(trustedRootCertificate2);

validator.validateChain(certificateChain);

X509Certificate rootCertificate = validator.getTrustedRoot();
```

Example 5-1: Certificate validation

The certificate validation gets current CRLs or OCSP responses automatically, by default. In certain use cases, however, it is required that the application can pass archived CRLs to the validation method because there is no standardized mechanism for retrieving old CRLs. This is especially useful for validating certificates with respect to a time in the past, which is usually required for validating archived signatures or archived signed documents, e.g. signed e-mails or signed legal documents. The application can provide multiple CRLs to allow for revocation checking all certificates in the certificate chain. As additional parameter, the application passes the validation time to the validation method. This is the time when the end-entity certificate was used, i.e. the application needs to know if the certificate was valid at this time. If the application does not provide a time or sets it to `null`, the validation method takes the current time for validation.

```
X509Certificate trustedRoot = ...
CertValidator validator = new PkixCertValidator();
validator.addTrustedCertificate(trustedRoot);

X509CRL caCrl = ...
X509CRL endUserCrl = ...
validator.addCRL(caCrl);
validator.addCRL(endUserCrl);

X509Certificate[] certificateChain = ...
Date validationDate = ...
validator.validateChain(certificateChain, validationDate);
```

Example 5-2: Certificate validation with a time in the past

6 References

[CADES]	"Electronic Signatures and Infrastructures (ESI); CMS Advanced Electronic Signatures (CAdES)", ETSI TS 101 733, version 1.7.3, January 2007, Sophia Antipolis Cedex, France, available online at http://www.etsi.org
[CMS]	Housley R., "Cryptographic Message Syntax (CMS)", RFC 3852, IETF, Network Working Group, July 2004, available online at http://www.ietf.org/rfc/rfc3852.txt
[ECC-CMS]	Sean Turner, Dan Brown: "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)" RFC 5753, IETF, Network Working Group, June 2009, available online at http://datatracker.ietf.org/doc/rfc5753/
[ECRYPT]	"ECRYPT Yearly Report on Algorithms and Keysizes (2006) ", Revision 1.1, European Network of Excellence in Cryptology, 29. January 2007, available online at http://www.ecrypt.eu.org
[ESSCERT]	Schaad J., "Enhanced Security Services (ESS) Update: Adding CertID Algorithm Agility", RFC 5035, IETF, Network Working Group, August 2007, available online at http://www.ietf.org/rfc/rfc5035.txt
[SP 800-57]	Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid: "NIST Special Publication 800-57: Recommendation for Key Management – Part 1: General" (Revised), March 2007, available online at http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf
[PKIX]	Housley R., Polk W., Ford W., Solo D., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Pro-file", RFC 3280, IETF, Network Working Group, April 2002, available online at http://www.ietf.org/rfc/rfc3280.txt
[SMIME]	Ramsdell B., " Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification", RFC 3851, IETF, Network Working Group, July 2004, available online at http://www.ietf.org/rfc/rfc3851.txt
[XADES]	"XML Advanced Electronic Signatures (XAdES)", ETSI TS 101 903, version 1.3.2, March 2006, Sophia Antipolis Cedex, France, available online at http://www.etsi.org
[XML]	"Extensible Markup Language (XML) 1.0", 4th edition, W3C Recommendation, the W3C, 16 August 2006, available online at http://www.w3.org/TR/2006/REC-xml-20060816/
[XMLDSIG]	"XML-Signature Syntax and Processing", W3C Recommendation, the W3C, 12 February 2002, available online at http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/