JDigiDoc Programmer’s Guide

Document Version: 3.11

Library Version: 3.11

Last update: 29.06.2015

1. Document versions

Document information	
Created on	22.01.2013
Reference	JDigiDoc Programmer's Guide
Receiver	Sertifitseerimiskeskus AS
Author	Veiko Sinivee, Kersti Üts, Kristi Uukkivi
Version	3.11

Version information		
Date	**Version**	**Changes**
	2.7	The latest version of "JdigiDoc Programmer's Guide" created by Veiko Sinivee
30.09.2011	2.701	Initial draft by KnowIT for the new version based on v2.7, with following changes and additions: - under Introduction: added general overview info about the document contents, DigiDoc framework, security model and digitally signed file formats - under Overview: updated lists for References, Terms and acronyms and Dependencies/Environment (adding Apache Ant, Java Mail and JCE Unrestricted Policy Files) - under JdigiDoc architecture: added package diagram, included ee.sk.xmlenc, ee.sk.xmlenc.factory, ee.sk.digidoc.c14n and ee.sk.digidoc.tsl to the JdigiDoc packages overview list - under JdigiDoc utility: added general info on usage, configuration options, list of commands; added detailed explanation to main commands and command line parameters; added sample use cases for commonly used tasks with the utility tool - under JdigiDoc testing: added list of currently supported tokens and CA's; testing results for XadES Remote Plugtests and sample testing procedures for cross-usability - renewed overall document formatting and styles, based on SK's templates
24.10.2011	2.702	Revised according to developer feedback; additional information added for configuration entries, national solutions and cross-border support.
30.12.2011	2.7.1	Revised API description and PKCS12 support.
20.02.2012	3.6	Updated to 3.6 version
22.05.2012	3.6.1	Revised environment, configuration, certificates' usage and JdigiDoc utility program's description

22.01.2013	3.7	Updated to 3.7 version: removed EMBEDDED content type support, added description of signature verification settings, improved description of configuration file usage. Updated information about supporting older DigiDoc formats.
18.07.2013	3.8	Updated according to changes in 3.7.1 and 3.8 versions of the library: added information about verification warnings (chap. 5.1.7); added BDOC 2.0 signature format's support (removed BDOC 1.0 support); added ECC signature algorithm support. Changed the structure of the document for better readability, added chapter 8 describing JDigiDoc library's implementation notes.
11.12.2013	3.8.0.1	Updated signed file's validation process for API users (added chap. 5.2), updated BDOC 2.1 format's nonce calculation method, updated signer role handling, removed –list command from utility program.
13.12.2013	3.8.0.2	Fixed description of adding data file to container from a byte stream (method setBody). Added description of additional setting signature profile before signing if the container's content has been read in from input stream (chap. 5.1.4).
27.02.2014	3.8.1	Updated according to changes in version 3.8.1 of the library. Updated description of adding data files to DDOC/BDOC containers from memory, including changes to setBody(byte[]) method (chap. 5.1.3). Added jdigidoc.java utility program's command –ddoc-add-mem. Updated DigiDoc file's validation process (chap. 5.2). Removed Mobile-ID support.
24.08.2014	3.9	Updated according to changes in v3.9 of the library. Refactored chapter 5.3 "Encryption and decryption"
16.09.2014	3.9	Updated BDOC 2.1 container specific implementation notes, in JdigiDoc library, the signatures*.xml file's sequence number is counted from one.
29.01.2015	3.10	Updated according to changes in v3.10 of the library.
05.03.2015	3.10	Updated information in case of direct encryption/decryption vs using an intermediary DDOC container
29.06.2015	3.11	Updated description of checking for test signatures in chap. 5.2.3.1. Updated test OCSP responder address.

Table of contents

2. Introduction

This document describes JDigiDoc - a Java library of the Estonian ID-software. It is a basic building tool for creating applications handling digital signatures and their verification, encrypting and decrypting data. The digitally signed files are created in „DigiDoc format" (with .ddoc or .bdoc file extensions), compliant to XML Advanced Electronic Signatures (XAdES) [4], technical standard published by European Telecommunication Standards Institute (ETSI). JDigiDoc is also capable of encrypting/decrypting files (signed or unsigned), according to W3C XML Encryption Recommendation (XML-ENC).

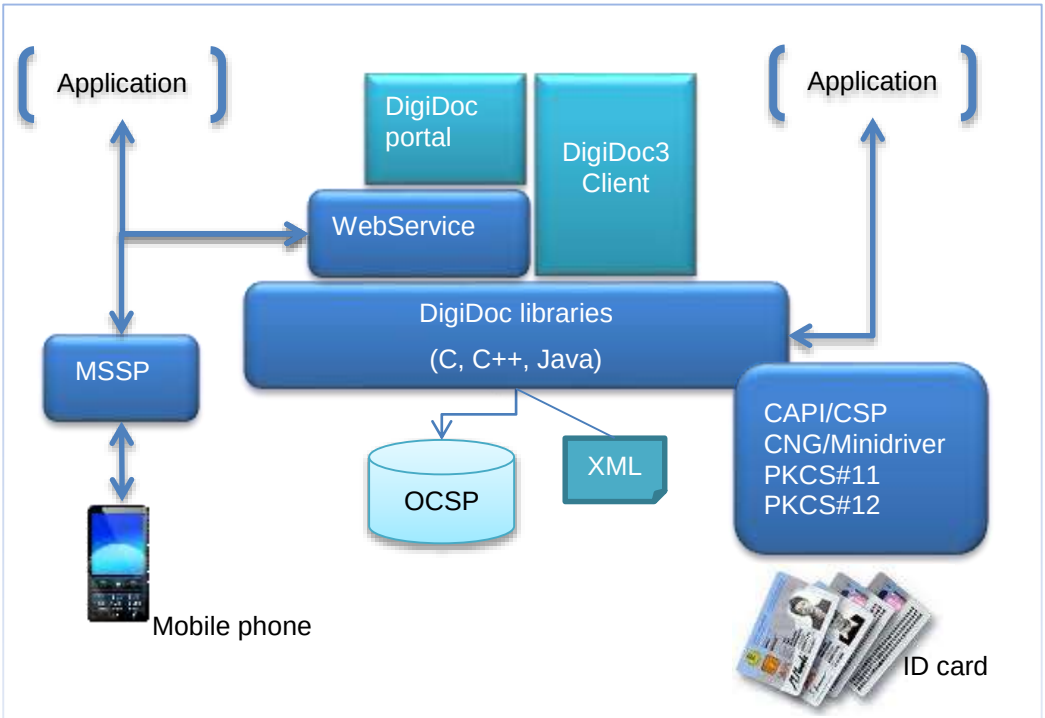
Development of the library can be monitored in GitHub environment: https://github.com/open-eid/jdigidoc.

This document covers the following information about JDigiDoc:

- Section 2 introduces the Estonian ID-software, its general security model and formats available for digitally signed files.
- Section 3 gives an overview of the functional properties supported by JDigiDoc library, describes the library's architecture and system requirements for using JDigiDoc.
- Section 4 describes JDigiDoc library's configuration possibilities.
- Section 5 explains the library's API for some of the most commonly used document signing and encryption tasks.
- Section 6 explains using the command line utility program for JDigiDoc, including sample use cases.
- Section 7 covers the currently supported tokens and CA's which have been tested with JDigiDoc, the current status of cross-border support for JDigiDoc and interoperability testing of JDigiDoc library's functionality.
- Section 8 gives an overview of JDigiDoc library's implementation notes which provide information about specific features of digitally signed files that are not defined in standards or specification documents but are implemented in JDigiDoc library.
- Appendix 1 provides a sample JDigiDoc.cfg configuration file.
- Appendix 2 describes different digital signature types that can be created with JDigiDoc library.

2.1 About DigiDoc

JDigiDoc library forms a part of the wider ID-software system framework which offers a full-scale architecture for digital signature and documents, consisting of software libraries (C, C++ and Java) [20], web service [15] and end-user applications such as DigiDoc Portal and DigiDoc3 Client according to the following figure:



1. DigiDoc framework

It is easy to integrate DigiDoc components into existing applications in order to allow for creation, handling, forwarding and verification of digital signatures and support file encryption/decryption. All applications share a common digitally signed file formats (current versions are DIGIDOC-XML 1.3 [1] and BDOC 2.1 [2]).

2.2 Format of digitally signed file

Actively used digitally signed file formats in ID-software system are:

- **BDOC 2.1** – new format, recommended for new signatures, described in [2].
- **DIGIDOC-XML 1.3** - described in [1];

More information of the life cycle of digitally signed file formats can be found from http://www.id.ee/?lang=en&id=36161. Transformation to BDOC 2.1 format is described in http://www.id.ee/?id=34336.

File extension **.bdoc** or **.ddoc** is used to distinguish digitally signed files according to the described file formats.

Other historical formats that were used previously are SK-XML, DIGIDOC-XML 1.1, DIGIDOC-XML 1.2 and BDOC 1.0.

The format of the digitally signed file is based on **ETSI TS 101 903** standard called **XML Advanced Electronic Signatures (XAdES)** [4]. The XAdES standard defines formats for

advanced electronic signatures that remain valid over long periods and incorporate additional useful information in common use cases (like indication of the role or resolution of the signatory). DIGIDOC-XML 1.3 and BDOC 2.1 file formats define a subset of this standard.

In order to verify whether the signer's certificate was valid at the (claimed) time of signing, "**time-marks**" are used in addition to "time-stamps". **In case of JDigiDoc library, only the time-marking profile is supported** (see also chap. 2.2.4), signing (and certificate validation) time comes with OCSP response.

Time-marking profile enables the following features:

- allows for incorporating following signed properties
  - Certificate used for signing,
  - Claimed signing time (the signer's computer time),
  - Signature production place (optional),
  - Signer role or resolution (optional).
- incorporates full signer certificate's validity information within the signature
  - OCSP response (providing trusted signing time and signer certificate's validity confirmation),
  - OCSP responder certificate.

As a result, it is possible to verify signature validity without any additional external information – the verifier should trust the issuer of signer's certificate and the OCSP responder's certificate.

2.2.1 BDOC 2.1 file format

The format of the BDOC 2.1 digitally signed file is in compliance with the ETSI standard TS 103 171 [8] which further profiles the XAdES signature by putting limitations on choices.

The **BDOC Basic Profile (EPES profile)** is an XML structure containing a single cryptographic signature over the well-defined set of data. It does not contain any validation data for full signature validation such as timestamps or certificate validity confirmations. The profile is based on XAdES-EPES (Explicit Policy based Electronic Signature, see [4]).

In case of **BDOC with time-marks (TM profile)**, the proof of validity of the signer's certificate is added to the signature, the validity confirmation is obtained by using OCSP protocol (see also chap. 2.2.4). The BDOC TM profile is compliant to XAdES LT-Level requirements. The OCSP request's "nonce" field is a DER-encoding of the following ASN.1 data structure:[1]

```
TBSDocumentDigest ::= SEQUENCE {
       algorithm AlgorithmIdentifier,
       digest OCTET STRING
}
```

The element `digest` is a hash value of the binary value of the signature and the element `algorithm` determines used hash algorithm defined in RFC 5280 ([14]) clause 4.1.1.2.

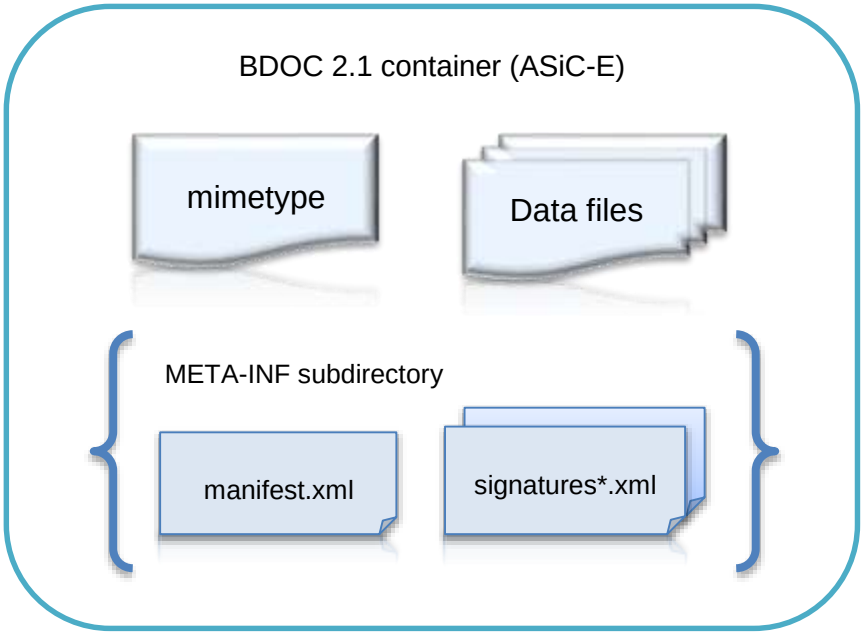
In case of BDOC 2.1 file format, the original data files (which were signed) along with the signatures are encapsulated within a ZIP container which is based on ETSI standard TS 102 918 [6] called **Associated Signature Containers (ASiC)**. The ETSI TS 103 174 [9] profiles in further on. The container type used in case of BDOC 2.1 documents is **Associated Signature Extended form (ASiC-E)**.

[1] Note: OCSP nonce field's value is calculated differently in case of DIGIDOC-XML 1.3 and BDOC 2.1 formats. See the specification documents of these file formats for more information.

ASiC-E container is a ZIP file consisting of the following objects:

- a file named "**mimetype**", containing only the following value: application/vnd.etsi.asic-e+zip
- **data files** in original format.
- **META-INF** subdirectory, consisting of:
  - **manifest.xml** – a file containing list of all folders and files in the container. The list does not contain the "mimetype" file and files in META-INF subdirectory.
  - **signatures*.xml** – one file for each signature, '*' in the file's name denotes the sequence number of a signature (counting starts from zero). The signatures*.xml file also incorporates certificates, validity confirmation and meta-data about the signer.

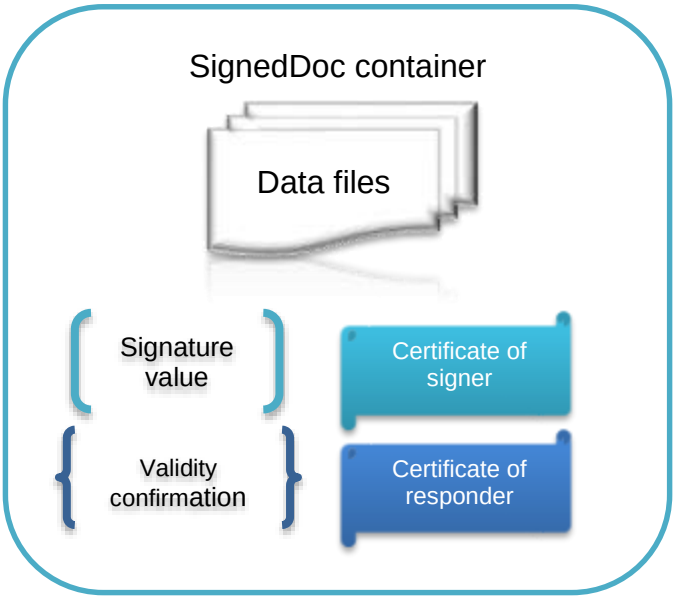
When BDOC 2.1 container is signed then all files in the container are signed, except of the mimetype file and files in META-INF subdirectory.



2. BDOC 2.1 container's contents

2.2.2 DIGIDOC-XML 1.3 file format

In case of DIGIDOC-XML 1.3 file format, the original data files (which were signed) along with the signature(s), validation confirmation(s) and certificates are encapsulated within a single XML container with <SignedDoc> as a root element.



3. SignedDoc container

2.2.3 Comparison of BDOC 2.1 and DIGIDOC-XML 1.3 implementations

The following table gives an overview of the main differences between DIGIDOC-XML 1.3 and BDOC 2.1 features that are implemented in JDigiDoc.

Note: the implemented features do not completely cover all the features that are described in the specification documents. See chapter "8 JDigiDoc library's implementation notes" for more information.

Feature	DIGIDOC-XML 1.3 (.ddoc)	BDOC 2.1 (.bdoc)
Container Format	Single XML file	Zip-file
Data file adding mode	- EMBEDDED_BASE64 (embeds binary data in base64 encoding) **Note**: EMBEDDED (embeds pure text or XML and DETACHED (adds only reference to an external file) data file adding modes are no longer supported	- BINARY
Contents of the container	- Data files embedded in the single ddoc XML file in base64 encoding - Signatures embedded in the single ddoc XML file	- Data files in original format - mimetype file with mime-type of the container - one META-INF/signatures*.xml file for each signature - META-INF/manifest.xml file with meta-data

Currently supported profiles	N/A **Note:** signatures added to DDOC documents are analogous to BDOC signatures with TM profile.	**EPES** - signature without time-mark, file can be created but gives validation error. Not tested periodically in JDigiDoc. **TM** – (time-mark) the default profile, tested periodically. **Note1**: due to historical reasons, the EPES profile is referred to as BES in JDigiDoc. **Note 2**: TS profile (using time-stamps) are not supported.
Supported data file digest type	Only **SHA-1** (set automatically)	Supported and tested digest type is **SHA-256**
Supported signature value digest type	Only **SHA-1** (set automatically)	Supported and tested digest types: **SHA-224** **SHA-256**
Supported signature algorithm	RSA	RSA ECDSA

2.2.4 DigiDoc time-marking security model

The general security model of the time-marking mechanism used in ID-software works by obtaining proof of validity of the signer's X.509 digital certificate issued by a certificate authority (CA) at the time of signature creation.

This proof (also named as "time-mark") is obtained in the format of Online Certificate Status Protocol (OCSP, [7]) response. Also, the hash of the created signature is sent within the OCSP request and received back within the response. This allows interpreting of the positive OCSP response as "at the time I saw this digitally signed file, corresponding certificate was valid", meaning that the OCSP response gives proof for the signer certificate's validity and also proof of the time when the signature existed. Thus, the time of issuing the OCSP response is interpreted as trusted signature creation time.

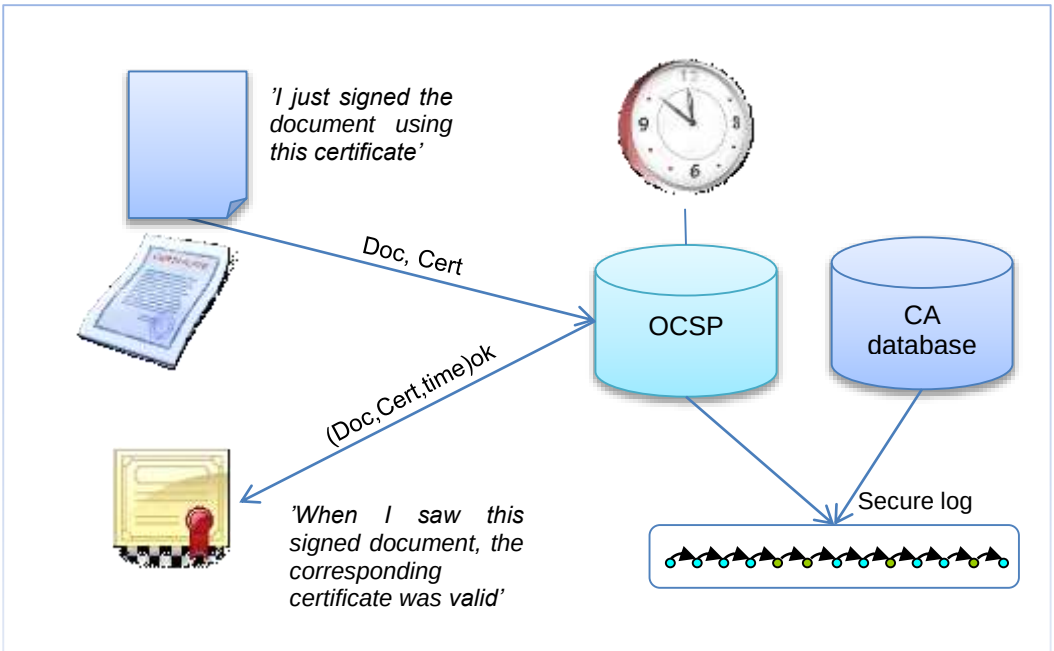
The OCSP response is stored within the signed document. This allows the signing time and signer certificate's validity to be validated later on, even after the signer's certificate has become invalid.

The OCSP service is acting as a digital e-notary confirming signatures created locally with a smart card. From infrastructure side, this security model requires a standard OCSP responder. Hash of the signature is placed on the "nonce" field of the OCSP request structure. In order to achieve the freshest certificate validity information, it is recommended to run the OCSP responder in "real-time" mode meaning that:

- certificate validity information is obtained from live database rather than from CRL (Certificate Revocation List)
- the time value in the OCSP response is actual (as precise as possible)

To achieve long-time validity of digital signatures, a secure log system is employed within the model. All OCSP responses and changes in certificate validity are securely logged to preserve digital signature validity even after private key compromise of CA or OCSP responder. It is important to notice that additional time-stamps are not necessary when employing the security model described:

- time of signing and time of obtaining validity information is indicated in the OCSP response
- the secure log provides for long-time validity without need for archival timestamps



4. DigiDoc security model

3. Overview

The current chapter gives an overview of **JDigiDoc** software library by describing the supported functionality, the general architecture and JDigiDoc library's dependencies.

3.1 References and additional resources

[1] DIGIDOC-XML 1.3	DigiDoc format specification, version 1.3.0 http://id.ee/public/DigiDoc_format_1.3.pdf
[2] BDOC2.1:2013	BDOC – Format for Digital Signatures. Version 2.1:2013 https://www.sk.ee/repository/bdoc-spec21.pdf http://id.ee/public/bdoc-spec21-est.pdf
[3] XML-DSIG	IETF RFC 3275: XML-Signature Syntax and Processing http://www.ietf.org/rfc/rfc3275.txt
[4] XAdES	ETSI TS 101 903 V1.4.2 (2010-12) – XML Advanced Electronic Signatures http://www.etsi.org/deliver/etsi_ts/101900_101999/101903/01.04.02_60/ts_101903v010402p.pdf

[5] OpenDocument	OASIS "Open Document Format for Office Applications. Version 1.2 Part 3: Packages" http://docs.oasis-open.org/office/v1.2/cs01/OpenDocument-v1.2-cs01-part3.html#__RefHeading__752803_826425813
[6] ASiC	ETSI TS 102 918 V1.2.1 (2012-02) - Associated Signature Containers http://www.etsi.org/deliver/etsi_ts/102900_102999/102918/01.02.01_60/ts_102918v010201p.pdf
[7] RFC6960	X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP http://tools.ietf.org/html/rfc6960
[8] XAdES Baseline Profile	ETSI TS 103 171 V2.1.1 (2012-03) http://www.etsi.org/deliver/etsi_ts/103100_103199/103171/02.01.01_60/ts_103171v020101p.pdf
[9] ASiC Baseline Profile	ETSI TS 103 174 V2.1.1 (2012-03) http://www.etsi.org/deliver/etsi_ts/103100_103199/103174/02.01.01_60/ts_103174v020101p.pdf
[10] DSA	Estonian Digital Signatures Act http://www.legaltext.ee/et/andmebaas/tekst.asp?loc=text&dok=X30081K6&keel=en&pg=1&ptyyp=RT&tyyp=X&query=digitaalallkirja
[11] XML-ENC	http://www.w3.org/TR/xmlenc-core/
[12] CDOC 1.0	Encrypted DigiDoc Format Specification http://id.ee/public/SK-CDOC-1.0-20120625_EN.pdf
[13] TSL	ETSI TS 102 231 ver. 3.1.2 (2009-12) - Electronic Signatures and Infrastructures (ESI); Provision of harmonized Trust-service status information
[14] RFC 5280	Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile http://tools.ietf.org/html/rfc5280
[15] DigiDocService Specification	EN: http://sk.ee/upload/files/DigiDocService_spec_eng.pdf ET: http://www.sk.ee/upload/files/DigiDocService_spec_est.pdf
[16] X.509 V3 Certificate Profile	ETSI TS 102 280 (V1.1.1) - X.509 V3 Certificate Profile for Certificates Issued to Natural Persons http://www.etsi.org/deliver/etsi_ts/102200_102299/102280/01.01.01_60/ts_102280v010101p.pdf
[17] ESTEID profile	Certificates on identity card of Republic of Estonia, version 3.3 https://sk.ee/upload/files/ESTEID_profiil_en-3_3.pdf

[18] Institution certificate profile	Profile of institution certificates and Certificate Revocation Lists, version 1.3 https://sk.ee/upload/files/SK_Profile%20of%20institution%20certificates%20and%20Revocation%20List.pdf
[19] JDigiDoc release notes	JDigiDoc library's release notes http://id.ee/index.php?id=35783
[20] DigiDoc libraries	http://id.ee/index.php?id=30486
[21] ID-software GitHub project	https://github.com/open-eid
[22] JDigiDoc GitHub repository	https://github.com/open-eid/jdigidoc

3.2 Terms and acronyms

ASiC	Associated Signature Containers
ASiC-E	Extended Associated Signature Containers. A type of ASiC container.
BDOC 2.1 (.bdoc)	Term is used to denote a digitally signed file format which is a profile of XAdES and follows container packaging rules based on OpenDocument and ASiC standards. The document format has been defined in [2].
BES	Basic Electronic Signature (XAdES-BES). A form of XAdES signature which does not incorporate any validation data, i.e. OCSP confirmation (time-mark) or time-stamp has not been added to the signature. **Note**: for historical reasons, in JDigiDoc library's source code, the term "BES" is used to denote both BES form and EPES form of a signature. The signature form is also referred to as "technical signature" in the context of the current document. In BDOC 2.1 specification, BES signature is used as a base form for qualified signature with a time-stamp (see [2], chap 6.2; time-stamps are currently not supported in JDigiDoc library). In DIGIDOC-XML 1.3 (see [1]), BES signature is used as a base form for a qualified signature with time-mark. **Note**: in the context of Estonian legislation, this form of a signature is not qualified electronic signature and is not equivalent to a handwritten signature.
CDOC (.cdoc)	The term denotes a format of an encrypted DigiDoc document that is based on XML-ENC profile. The document format has been defined in [12].

CRL	Certificate Revocation List, a list of certificates (or more specifically, a list of serial numbers for certificates) that have been revoked, and therefore should not be relied upon.
DIGIDOC-XML (.ddoc)	The term is used to denote a DigiDoc document format that is based on the XAdES standard and is a profile of that standard. The document format has been defined in [1]. The profile does not exactly match any subsets described in XAdES standard – the best format name would be "XAdES-C-L" indicating that all certificates and OCSP confirmations are present but there are no "pure" timestamps. A DIGIDOC-XML file is basically a <SignedDoc /> container that contains original data files and signatures. The file extension for DIGIDOC-XML file format is ".ddoc", MIME-type is "application/ddoc".
EPES	Explicit Policy based Electronic Signature (XAdES-EPES). A form of XAdES signature, similar to BES signature form but has an additional element <SignaturePolicyIdentifier>. **Note**: for historical reasons, in JDigiDoc library's source code, the term "BES" is used to denote both BES form and EPES form of a signature. The signature form is also referred to as "technical signature" in the context of the current document. In BDOC 2.1 specification, EPES signature is used as a base form for qualified signature with a time-mark (see [2], chap 6.1). **Note**: in the context of Estonian legislation, this form of a signature is not qualified electronic signature and is not equivalent to a handwritten signature.
OCSP	Online Certificate Status Protocol, an Internet protocol used for obtaining the revocation status of an X.509 digital certificate
OCSP Responder	OCSP Server, maintains a store of CA-published CRLs and an up-to-date list of valid and invalid certificates. After the OCSP responder receives a validation request (typically an HTTP or HTTPS transmission), the OCSP responder either validates the status of the certificate using its own authentication database or calls upon the OCSP responder that originally issued the certificate to validate the request. After formulating a response, the OCSP responder returns the signed response, and the original certificate is either approved or rejected, based on whether or not the OCSP responder validates the certificate.
SK	AS Sertifitseerimiskeskus (Certification Centre Ltd.). Certificate Authority in Estonia
X.509	an ITU-T standard for a public key infrastructure (PKI) and Privilege Management Infrastructure (PMI) which specifies standard formats for

	public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm
XAdES	XML Advanced Electronic Signatures, a set of extensions to XML-DSig recommendation making it suitable for advanced electronic signature. Specifies precise profiles of XML-DSig for use with advanced electronic signature in the meaning of European Union Directive 1999/93/EC.
XML-DSIG	a general framework for digitally signing documents, defines an XML syntax for digital signatures and is defined in the W3C recommendation XML Signature Syntax and Processing

3.3 Supported functional properties

JDigiDoc is a library of Java classes offering the following functionality:

- **creating** containers in supported formats and adding data files;
- **signing** DigiDoc documents using smart cards or other supported cryptographic tokens;
- adding **time marks** and **validity confirmations** to digital signatures using OCSP protocol;
- **validating** the digital signatures;
- **extracting** data files from DigiDoc document;
- digital **encryption and decryption** of data files.

The following table describes functional features that are supported with JDigiDoc.

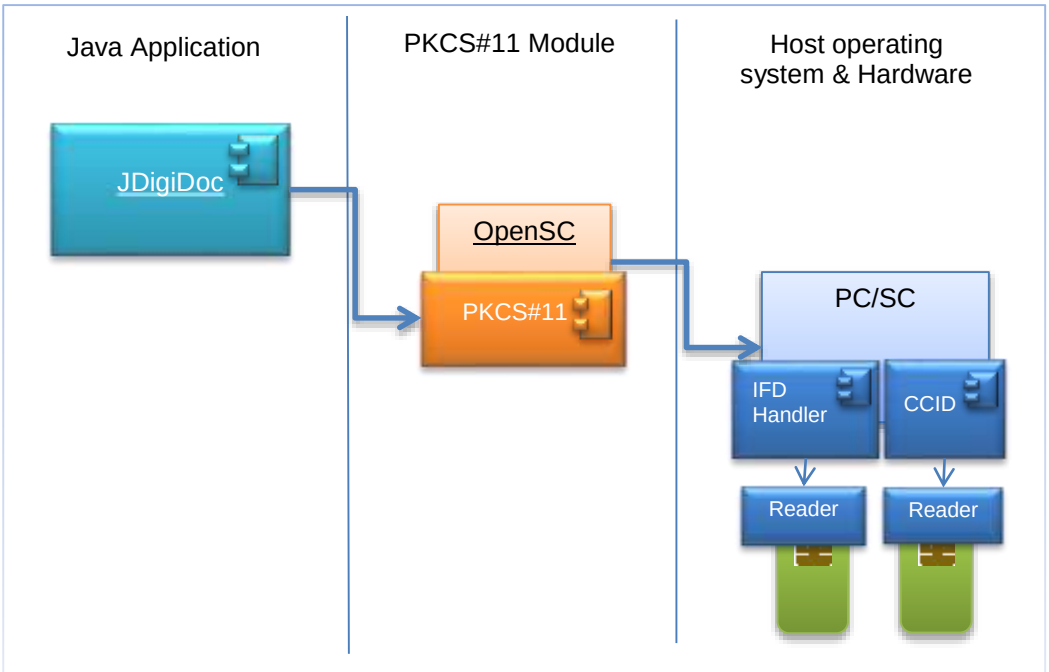
Feature	Supported values
Signed DigiDoc document format	- **BDOC 2.1 –** newer digital signature format, recommended for new signatures, described in [2]. - **DIGIDOC-XML 1.3** – digital signature format, described in [1]. **Note**: older DigiDoc file formats SK-XML, DIGIDOC-XML 1.1 and DIGIDOC-XML 1.2 are supported only for backward compatibility in case of digital signature verification and data file extraction operations (creating new files and modifying existing files is not supported). **Note**: BDOC 1.0 file format is not supported (files in this format are recognized by the library but handling the files is not supported).

Signature profile	Signature profiles are based on the profiles defined by XAdES ([4]). - **TM (time-mark) -** the default profile, actual certificates and revocation data are added to the signed document to allow verification in future even if their original source is not available; uses **time marking**. In case of BDOC 2.1 document format, the "SignaturePolicyIdentifier" element is mandatory (see also [2]). - **BES/EPES -** basic profile, doesn't provide long term validation. The signature doesn't contain any validation data like time-mark or certificate validity confirmation (OCSP response). The signature with BES or EPES profile is a technical signature and is expected to produce specific validation error messages (see also "Appendix 2: Signature types"). **Note1**: for historical reasons, in the current document and in JDigiDoc library, the BDOC 2.1 format's EPES profile is also referred to as BES. **Note2:** TS profile (using timestamps as described in BDOC 2.1 specification [2], chap. 6.2) is not supported in DigiDoc libraries.
Signature creation module	- **PKCS#11 –** the default module for singing with smart card (e.g. Estonian ID card or any other smartcard provided that you have the external native PKCS#11 driver for it). - **PKCS#12 –** module for signing with a software token. **Note:** support for M-ID signing via DigiDocService [15] in JDigiDoc is deprecated.
Cryptographic token type	- **Smart card,** e.g. Estonian ID card. Supported signature creation module is PKCS#11. - **Software token -** a PKCS#12 container (.p12 or .pfx) file which includes a certificate and accompanying public and private keys in a single file. The private key is protected with a password-based symmetric key. The token is named "software token" as it is stored in the file system and not on a smartcard or other physical cryptographic device. Supported signature creation module is PKCS#12. Note that the signature that is created with a software token is a technical signature and is expected to produce verification error messages (see also "Appendix 2: Signature types"). - **USB cryptostick** - Aladdin eToken device. Supported signature creation module is PKCS#11.
Public-key algorithm	- **RSA** - **ECDSA -** support for ECDSA algorithm has only been tested with 256 bit keys prime256v1(secp256r1). Testing has been carried out with PKCS#12 software tokens (via PKCS#12 signature creation module in JDigiDoc utility program).
Encrypted document format	- **XML-ENC 1.0**

Further information about specific functional features that are not strictly defined in specification documents but are implemented in JDigiDoc library can be found from chapter "8 JDigiDoc library's implementation notes".

3.4 Component model

The figure below describes the architecture of software and hardware components that are used when creating signatures with JDigiDoc library.



5. Sample JDigiDoc implementation using PKCS#11/ smart cards for digital signing

Component	Description
OpenSC	Set of libraries and utilities to work with smart cards, implementing PKCS#11
PKCS#11	Widely adopted platform-independent API to cryptographic tokens (HSMs and smart cards), a standard management module of the smart card and its certificates
PC/SC	Standard communication interface between the computer and the smart card, a cross-platform API for accessing smart card readers
IFDHandler	Interface Device Handler for CCID readers
CCID	USB driver for Chip/Smart Card Interface Devices
Reader	Device used for communication with a smart card

3.5 JDigiDoc architecture

JDigiDoc library consists of the following packages:

ee.sk.digidoc – Core classes of JDigiDoc modeling the structure of various XML-DSIG and XAdES entities. DigiDocException class includes the error codes that are used in the library.

- ee.sk.digidoc.factory – Exchangeable modules implementing various functionality that you might wish to modify and interfaces to those modules

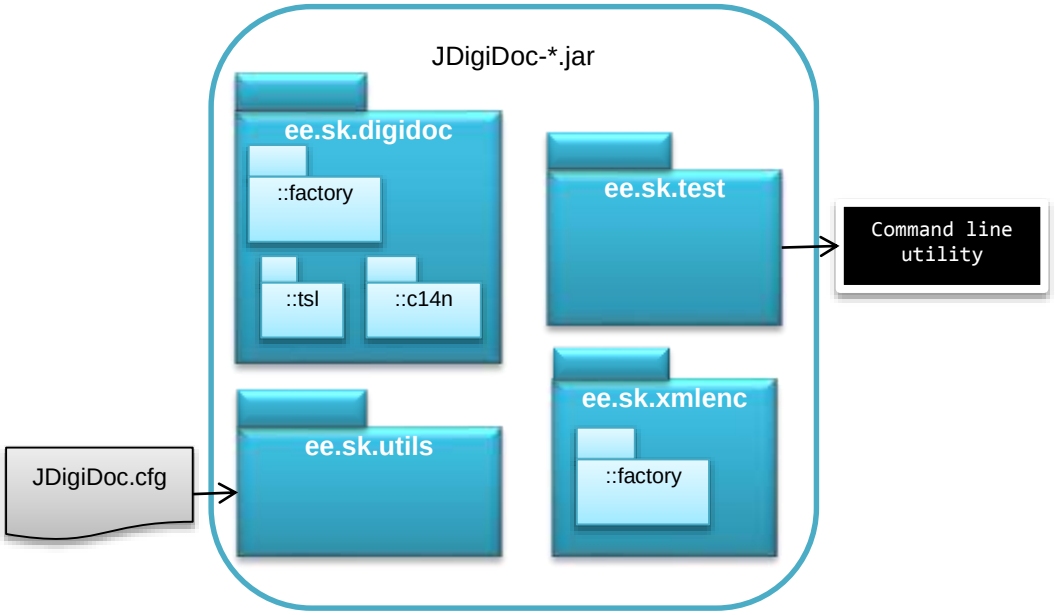
- ee.sk.digidoc.c14n – Classes for XML canonicalization implementation with TinyXMLCanonicalizer
  - ee.sk.digidoc.c14n.common – Additional classes for TinyXMLCanonicalizer implementation
- ee.sk.digidoc.tsl – Classes modeling the ETSI TS 102 231 V3.1.1. Trust Service Status List types. **Note:** the Trust Service Status List (TSL) functionality is currently not fully supported with JDigiDoc.

ee.sk.utils – Configuration and other utility classes

ee.sk.test – Sample and command line utility programs

ee.sk.xmlenc – Classes modeling XML entities specified in XML-ENC standard

- ee.sk.xmlenc.factory – Classes for parsers of encrypted files



6. JDigiDoc packages

For additional information about the JDigiDoc library's classes and their contents, see the full API description (javadoc) that is included in the JDigiDoc library's distribution package.

3.6 Dependencies

JDigiDoc depends on a number of libraries, some of which are base components and others which depend on the base modules that have been used.

Base Component	Description
Java Platform	JDK/JRE 1.5 or newer **Note**: currently JDK/JRE versions 1.6 and 1.7 are included in testing
Bouncy-Castle cryptographic library	Used in cryptographic operations. This library was chosen as it is a freeware module.

TinyXMLCanonicalizer	ee.sk.digidoc.c14n. TinyXMLCanonicalizer is a small and very efficient XML canonicalizer with a small memory footprint and no further dependencies. Good enough for basic ddoc/bdoc usage, but has some problems with XML namespaces and special symbols handling
Jakarta Log4j	Required for the Apache XML Security library and by JDigiDoc itself for logging purposes
Apache Commons Codec	Required for Base64 encoding
Apache Commons Compress	Required for using BDOC format with utf-8 encoding

3.7 Environment

The following libraries need to be added to the CLASSPATH environment variable in order to use JDigiDoc:

Library	Description
JDigiDoc-*.jar	JDigiDoc library itself ('*' denotes the library's version number)
bcmail-jdk*-151.jar **bcprov-jdk*-151.jar** **bcpkix-jdk*-151.jar** **or newer**	Bouncy-Castle cryptographic library, a Java implementation of cryptographic algorithms. Choose the releases according to your JDK version, e.g. for JDK 5.0 → bcmail-jdk15-151.jar, etc. Latest releases available from: http://www.bouncycastle.org/latest_releases.html
jakarta-log4j-1.2.6.jar or newer	Jakarta Log4j library Latest releases available from: http://logging.apache.org/log4j/1.2/download.html
commons-compress-1.3.jar	Apache Commons Compress library. Latest releases available from: http://commons.apache.org/compress/download_compress.cgi
commons-codec-1.6.jar	Apache Commons Codec library. Latest releases available from: http://commons.apache.org/codec/download_codec.cgi
iaikPkcs11Wrapper.jar • ***for Windows:*** + ***opensc-pkcs11.dll*** + ***Pkcs11Wrapper.dll***	If you want to create RSA-SHA1/SHA2 digital signatures via a smart card/card reader device and access the latter using a PKCS#11 driver, then add the library file: • iaikPkcs11Wrapper.jar to CLASSPATH environment variable. It is a set of Java classes and interfaces that reflects the PKCS#11 API. Additionally in Windows environment, you need to copy the following files: • opensc-pkcs11.dll

	• Pkcs11Wrapper.dll (either 64-bit and/or 32-bit version according to your JDK/JRE's version) to a directory listed in the PATH environment variable
• ***for Linux:*** + ***libopensc-pkcs11.so or opesc-pkcs11.so*** + ***libpkcs11wrapper.so*** • ***for OS X:*** + **libpkcs11wrapper.jnilib**	In Linux environment, copy the following shared object libraries: • libopensc-pkcs11.so or opesc-pkcs11.so • libpkcs11wrapper.so to the directory {JAVA_HOME}\jre\lib\i386
JCE Unlimited Strength Jurisdiction Policy Files: **local_policy.jar** **US_export_policy.jar**	By default, current versions of the JDK have a deliberate 128-bit key size restriction built in which throws an `InvalidKeyException`, with the message *"Illegal key size or default parameters"*. If you get this exception, you can remove the restriction by overriding the security policy files with others that Sun provides. The version of the policy files must be the same as the version of your JDK – e.g. for: JDK 5.0 -> JCE USPF 5 JDK 6.0 -> JCE USPF 6 Copy the JCE Unlimited Strength Jurisdiction Policy Files over the ones already in the standard JDK/JDRE directory (adjust pathname separators according to your environment): {JAVA_HOME}\lib\security . Note that if you are using Windows, the JDK install will normally install a JRE and a JDK in two separate places - generally both of these will need to have the new policy files installed in it. The JCE Unlimited Strength Jurisdiction Policy Files can be downloaded from: http://www.oracle.com/technetwork/java/javasebusiness/downloads/index.html
esteidtestcerts.jar	Estonian test certificate package, should be added to the CLASSPATH when validating or creating signatures with test certificates. Accessible from: https://installer.id.ee/media/esteidtestcerts.jar **NB!** The package must be removed from classpath in case of live applications because JDigiDoc library does not give notifications to users in case of test signatures (except of the JDigiDoc utility program, since v3.8).

4. Configuring JDigiDoc

4.1 Loading configuration settings

JDigiDoc uses the class ee.sk.utils.ConfigManager for reading configuration data from a Java property file – **JDigiDoc.cfg**.

There are two alternative methods for loading configuration settings:

1. **ee.sk.utils.ConfigManager.init(String fileName)** method can be used for loading configuration settings by providing the configuration file's name. The fileName parameter can be either:
    - full path and name of the configuration file in file system or only the file's name if it is in a location referenced by CLASSPATH,
    - location of the configuration file in a jar container that has been added to CLASSPATH. For example, the default configuration file jdigidoc.cfg that is embedded in JDigiDoc -*.jar container (* indicates the version of the library) can be loaded by setting the fileName parameter to "jar://JDigiDoc.cfg".
    - URL value referring to the configuration file's name and location, for example "https://svn.eesti.ee/projektid/idkaart_public/trunk/jdigidoc/jdigidoc/src/main/resources/jdigidoc.cfg".

    Note that when calling out the method repeatedly then it only overwrites the configuration entries that were already present in the configuration file that was used in the first call of the method. Therefore, it would be more convenient to include all of the needed configuration settings in the first configuration file and overwrite any additional settings by loading a smaller file during the program's working time when necessary.

    For example, if you would like to load alternative CA configuration entries with parameter values that were not present in the initial configuration file then you need to use methods of ee.sk.digidoc.tsl.DigiDocTrustServiceFactory class to load the values (i.e. re-initialise the object with DigiDocTrustServiceFactory.init() method):

```
ConfigManager.instance().getTslFactory().init();
```

2. **ee.sk.utils.ConfigManager.init(Hashtable hProps)** method can be used to load configuration settings from previously initialised java.util.Properties object. The method is useful, for example, when loading configuration settings form a database or during working time of a service without the need for restart.

4.2 Configuration parameters

For a sample configuration file provided with JDigiDoc, see Appendix 1.

Below is an overview of the configuration file's main sections and entries. The following color notation is used for specific parameter values:

- ***bold*** for default values which do not usually need to be changed by the user
- purple for indicating values which should be checked and modified according to user
- # blue for listing possible alternatives, where applicable

Signature processor settings (exchangeable modules)

For replacing one of the standard modules with your implementation, you should place the module in CLASSPATH and register its class name here

Parameter	Comments
DIGIDOC_SIGN_IMPL	Module used for signature creation. ***ee.sk.digidoc.factory.PKCS11SignatureFactory*** (implementation module using smartcards over PKCS#11 driver and IAIK PKCS#11 wrapper library to access external native language PKCS#11 drivers). Not thread safe
DIGIDOC_NOTARY_IMPL	Module for notary functions. ***ee.sk.digidoc.factory.BouncyCastleNotaryFactory*** (implementation module for getting OCSP confirmations using BouncyCastle library)
DIGIDOC_FACTORY_IMPL	Module for reading and writing DigiDoc documents. ***ee.sk.digidoc.factory.SAXDigiDocFactory*** (implementation using a SAX parser)
DIGIDOC_TIMESTAMP_IMPL	Module for timestamp functions. ***ee.sk.digidoc.factory.BouncyCastleTimestampFactory*** (implementation using BouncyCastle library). **Note**: Implementation of time-stamp related functionality is experimental in JDigiDoc.
CANONICALIZATION_FACTORY_IMPL	Module for canonicalization functions. ***ee.sk.digidoc.c14n.TinyXMLCanonicalizer*** (implementation using Tiny XML Canonicalizer)
DIGIDOC_TSLFAC_IMPL	Module for TSL functions. ***ee.sk.digidoc.tsl.DigiDocTrustServiceFactory*** (implementation module using a SAX parser)
ENCRYPTED_DATA_PARSER_IMPL	Module for reading and writing small encrypted files. ***ee.sk.xmlenc.factory.EncryptedDataSAXParser*** (implementation using a SAX parser)
ENCRYPTED_STREAM_PARSER_IMPL	Module for reading and writing large encrypted files. ***ee.sk.xmlenc.factory.EncryptedStreamSAXParser*** (implementation using a SAX parser)

Security settings

Parameter	Description
DIGIDOC_SECURITY_PROVIDER	Security module used for cryptographic algorithms in Java. ***org.bouncycastle.jce.provider.BouncyCastleProvider***
DIGIDOC_SECURITY_PROVIDER_NAME	Name for the security provider. ***BC***

Big file handling

Parameter	Description
DIGIDOC_MAX_DATAFILE_CACHED	Maximum number of bytes per DataFile object to be cached in memory. If object size exceeds the limit then the data is stored in temporary file. If the parameter's value is set to a negative number then temporary files are not written to disk. ***4096*** *(4 KB)*
DIGIDOC_DF_CACHE_DIR	Specifies directory for storing temporary files. Default value is read from system parameter java.io.tmpdir # /tmp

Signature verification settings

Parameter	Description
CHECK_OCSP_NONCE	Used only in case of DIGIDOC-XML 1.3 files. Specifies if the OCSP response's nonce field's ASN.1 prefix is checked during signature creation and verification. By

	default, the value is set to false in order to support verification of .ddoc files created with JDigiDoc library's version below v3.7. The nonce field's ASN.1 prefix is always checked in case of BDOC 2.1 files. ***false*** # true

Note: The ASN.1 prefix specifies the digest algorithm that was used to calculate the nonce field's value, the prefix is mandatory according to RFC6960 specification (see also [7]). If the ASN.1 prefix is checked then it must contain the octet string ASN.1 identificator (0x04 0x14 in case of DDOC 1.3).

Default digest types for BDOC

According to a study on the use of cryptographic algorithms in state information systems published by the Estonian Department of State Information Systems in 2011, (http://www.riso.ee/et/files/kryptoalgoritmide_elutsykli_uuring.pdf, in Estonian), it's recommended to support and use hash functions belonging to at least the SHA-2 set – i.e. SHA-224, SHA-256, SHA-384 or SHA-512 in digital signing protocols.

In case of **BDOC 2.1** format, the default hash function to be used for new signatures and other digests is set in the JDigiDoc configuration file as **SHA-256**.

Parameter	Description
DIGIDOC_DIGEST_TYPE	Specifies the default digest type for all hash values in the BDOC 2.1 signature file (see additional notes below for signature value's digest) ***SHA-256***

Additional notes on default digest type usage:

- For the DIGIDOC-XML 1.3 format, SHA-1 will still the default digest type for all digests and cannot be altered.
- For BDOC 2.1 format, the default digest type is SHA-256 (except of the case which is described under the next point).
- For the BDOC 2.1 format and in case of Estonian ID cards with certificates issued before 2011, the SHA-224 digest type will be automatically selected and used for calculating signature value's digest (the final digest that is signed); other options are not being supported there. Note that other digest in the signature (e.g. data file digests, signer certificate's digest) are still calculated with SHA-256 (the default digest type).

Default profile

Profiles are based on the profiles defined by XAdES.

Parameter	Description
DIGIDOC_DEFAULT_PROFILE	Specifies the default profile to be used when creating a new document. **TM** - main signature profile to be used, using time- mark. # BES

Note: for historical reasons, in the current document and in JDigiDoc library, the BDOC 2.1 format's EPES profile is also referred to as BES.

PKCS#11 settings

If using the smart card over PKCS#11 module for creating signatures, then you must specify the following parameters according to your signature device here:

Parameter	Description
DIGIDOC_SIGN_PKCS11_DRIVER	Specifies the PKCS11 driver library used to communicate with the smart card. With Estonian ID cards for example, the following PKCS#11 libraries are used: **opensc-pkcs11.so *(used in Linux and OSX environment)*** *# opensc-pkcs11.dll (used in Windows environment)*
DIGIDOC_SIGN_PKCS11_WRAPPER	A wrapper library which makes PKCS#11 modules available from within Java. ***PKCS11Wrapper***

log4j configuration file

If you wish to replace the default log4j configuration file with your own or access it from a different location, please change the following parameter accordingly:

Parameter	Description
DIGIDOC_LOG4J_CONFIG	The location of the log4j.properties file **./log4j.properties**

OCSP responder settings

This DIGIDOC_OCSP_RESPONDER_URL setting applies to your default OCSP responder address when no other OCSP responder address for the CA is found in the OCSP responder data registered in your configuration file entries.

The default address provided (http://ocsp.sk.ee) is for the real-life OCSP Responder service to be used for Estonian ID cards.

Note: access to the real-life OCSP service (managed by Estonian CA – AS Sertifitseerimiskeskus, SK) is granted on the basis of contract (http://sk.ee/en/services/validity-confirmation-services). There are two ways of getting access to the OCSP service:

- accessing the service from specific IP address(es);
- by having an access certificate for accessing the service (i.e. sending OCSP request which are signed with the access certificate, see also the next section).

For testing purposes, use SK's test OCSP responder service available at http://demo.sk.ee/ocsp (previously http://www.openxades.org/cgi-bin/ocsp.cgi). For more information, please refer to http://www.id.ee/?lang=en&id=35755.

Parameter	Description
DIGIDOC_OCSP_RESPONDER_URL	OSPC Responder used if no other address is found for the CA in the locally registered entries. ***http://ocsp.sk.ee***
OCSP_TIMEOUT	Connect timeout in milliseconds; 0 means wait forever **30000**

Settings for signing OCSP requests or not

Whether you need to sign the OCSP requests sent to your OCSP responder or not depends on your responder.

Some OCSP servers require that the OCSP request is signed. To sign the OCSP request, you need to obtain and specify the certificates, which will be used for signing.

For example, accessing the SK's OCSP Responder service by private persons requires the requests to signed (limited access certificates can be obtained through registering for the service) whereas in case of companies/services, signing the request is not required if having a contract with SK and accessing the service from specific IP address(es) (IP-based access).

By default, this parameter value is set to FALSE – i.e. it is assumed that the user has IP-based access and the OCSP requests don't need to be signed. If the parameter is set to TRUE then you will also need to provide your access certificate's file location, password and serial number that have been issued to you for this purpose.

Parameter	Description
SIGN_OCSP_REQUESTS	Specifies if the OCSP requests will need to be signed or not ***FALSE*** ***# TRUE***
DIGIDOC_PKCS12_CONTAINER	Specifies your pkcs12 filename, e.g. ***./home/132936.p12d***
DIGIDOC_PKCS12_ PASSWD	Specifies your pkcs12 password, e.g. ***m15eTGpA***
DIGIDOC_ OCSP_SIGN_CERT_SERIAL	Specifies your pkcs12 certificate serial number e.g. ***129525***

You can use DigiDoc3 Client to find the access certificate's serial number, if you are using the same certificate which has been installed there – open "Settings" menu, "Server access certificate" tab, "Show certificate" button, and find the serial number under "Details" view.

CA certificates

The CA certificates registered in the configuration file will be used when creating and verifying signatures, in order to check the signer certificate's and OCSP responder certificate's trust hierarchy.

By default, the Estonian CA's certificates (both live and test certificates) have been registered in the JDigiDoc configuration file. The live CA and OCSP certificate files have been included in the JDigiDoc distribution but the test certificate files haven't. In order to use the test certificates, you need to copy the certificate files to a location referenced by the CLASSPATH (the files are accessible from https://installer.id.ee/media/esteidtestcerts.jar).

Note that if placing the certificates to some location referenced by the CLASSPATH, you can use **jar://** to get them (using forward slashes both on your Linux and other environments, e.g. jar://certs/TEST EECCRCA.crt)

Note: test certificates should not be used in live applications as the JDigiDoc library does not give notifications to the user in case of test signatures (even though since the library's version v3.8, the utility program displays test signature as a warning to the user). In case of live applications, the test certificates should be removed.

Parameter	Description
DIGIDOC_CAS	Number of 'local' CAs registered in your configuration file ***1***

DIGIDOC_CA_1_NAME … DIGIDOC_CA_*n*_NAME	Name of the registered CA's. The currently registered CA in JDigiDoc is: ***DIGIDOC_CA_1_NAME = AS Sertifitseerimiskeskus*** (The number of entries corresponds to DIGIDOC_CAS)
DIGIDOC_CA_1_TRADENAME … DIGIDOC_CA_*n*_TRADENAME	Trade name for the CA ***DIGIDOC_CA_1_TRADENAME = SK*** (The number of entries corresponds to DIGIDOC_CAS)
DIGIDOC_CA_1_CERTS … DIGIDOC_CA_*n*_ CERTS	Number of certificates for the specific CA. Currently, the following number of certificates is registered per CAs in JDigiDoc: ***DIGIDOC_CA_1_CERTS = 17*** (The number of entries corresponds to DIGIDOC_CAS)
DIGIDOC_CA_1_CERT1 … DIGIDOC_CA_*n*_CERT*n*,	Location of all certificates for each CA (The number of entries corresponds to each CA's DIGIDOC_CA_*_CERTS) **Note**: if the certificates' location has been referenced by the classpath, then you can enter jar:// for retrieving them, e.g. ***DIGIDOC_CA_1_CERT1 = jar://certs/EID-SK.crt***

OCSP responder certificates

When using the CAs registered in your configuration file and OCSP responses in signature creation and verification, you must provide the following details for each OCSP responder:

Parameter	Description
DIGIDOC_CA_1_OCSPS … DIGIDOC_CA_*n*_OCSPS	Number of OCSP responders for the specific CA. By default, only the OCSP responders for SK have been registered here, e.g. for SK: ***DIGIDOC_CA_1_OCSPS = 19***
DIGIDOC_CA_1_OCSP1_CA_CN … DIGIDOC_CA_*n*_OCSP *n* _CA_CN	Name of the CA for the specific OCSP responder being entered, e.g. ***KLASS3-SK***
DIGIDOC_CA_1_OCSP1_CA_CERT … DIGIDOC_CA_*n*_OCSP *n* _CA_CERT	Location of the CA's certificate for the specific OCSP responder being entered, e.g. ***jar://certs/KLASS3-SK.crt***
DIGIDOC_CA_1_OCSP1_CN … DIGIDOC_CA_*n*_OCSP*n*_CN	Name of the specific OCSP responder, e.g. ***KLASS3-SK OCSP RESPONDER***
DIGIDOC_CA_1_OCSP1_CERT … DIGIDOC_CA_*n*_OCSP*n*_CERT	Location of the OCSP responder's certificate, e.g. ***jar://certs/KLASS3-SK OCSP.crt***
DIGIDOC_CA_1_OCSP1_CERT_1 … DIGIDOC_CA_*n*_OCSP*n*_CERT_n	Specifies certificate(s) of the OCSP responder, e.g. ***jar://certs/KLASS3-SK OCSP 2006.crt***
DIGIDOC_CA_1_OCSP1_URL … DIGIDOC_CA_*n*_OCSP*n*_URL	Address for the OCSP responder, e.g. ***http://ocsp.sk.ee***

Registering or removing CAs and OCSP responders

For changing the CAs and certificate settings in JDigiDoc, new 'local' CAs, OCSP responders and certificates can be registered in the configuration file or already existing entries can be removed.

For example, for adding a new CA, the following parameters should be updated:

```
DIGIDOC_CAS     =       <add +1 if adding a new CA to the ones already registered>
…
DIGIDOC_CA_2_NAME       =       <enter name for a new CA, e.g. TEST2>
DIGIDOC_CA_2_TRADENAME  =       <enter short name for a new CA, e.g. T2>
DIGIDOC_CA_2_CERTS      =       <update or enter the number of CA certs, e.g. 3>
DIGIDOC_CA_2_CERT1      =       <enter locations for all the CA certs>
DIGIDOC_CA_2_CERT2      =       <e.g. jar://certs/CA_2_TESTcert2.crt>
DIGIDOC_CA_2_CERT3      =       <e.g. jar://certs/CA_2_TESTcert3.crt>
…
DIGIDOC_CA_2_OCSPS = <update or enter the number of OCSP responders for a CA, e.g. 1>
…
DIGIDOC_CA_2_OCSP1_CA_CN =<enter common name for CA, e.g. TEST2>
DIGIDOC_CA_2_OCSP1_CA_CERT =<enter cert for CA, e.g. jar://certs/CA_2_TESTcert2.crt>
DIGIDOC_CA_2_OCSP1_CN    =<enter common name for OCSP responder, e.g. TEST2
RESPONDER>
DIGIDOC_CA_2_OCSP1_CERT  =<enter cert for OCSP responder,
e.g.jar://certs/TEST2Responder.crt>
DIGIDOC_CA_2_OCSP1_URL   =<enter URL for OCSP responder, e.g. http://www.testOCSP.ee>
```

The newly registered CA and OCSP certificate files have to be copied to a location referenced by the CLASSPATH.

Note: If OCSP confirmations are to be used against certificates issued by any new CAs, then the necessary conditions set by the CA for accessing its OCSP service must be first met and the corresponding OCSP responder data then entered in the configuration file.

For removing a CA from the configuration file, all of the related entries should be deleted (both the CA and OCSP responder certificate data).

For removing only some certificates of a CA or its OCSP responders then delete the related entries from the configuration file. After removing an OCSP responder, update also the following parameter's value:

```
DIGIDOC_CA_*_OCSPS = <update the number of OCSP responders for the CA>
```

Log4j configuration file

JDigiDoc uses only a part of Apache XML Security library for XML canonicalization. Unfortunately this library requires putting references to DTD in one's XML documents and outputs lots of warnings if it doesn't find such references.

One way of discarding those warnings is to set the main logger in Log4j configuration file very restrictive and then selectively enable logging only for those components that you wish. For example:

Sample log4j.properties:

```
# root logger properties
log4j.rootLogger=FATAL, output, logfile

# JDigiDoc loggers
log4j.logger.ee.sk.utils.ConfigManager=INFO, output
log4j.logger.ee.sk.xmlenc.EncryptedData=INFO, output
log4j.logger.ee.sk.digidoc.DigiDocException=INFO, output
log4j.logger.ee.sk.digidoc.factory.PKCS11SignatureFactory=INFO, output
log4j.logger.ee.sk.digidoc.factory.SAXDigiDocFactory=INFO, output
log4j.logger.ee.sk.digidoc.factory.DigiDocVerifyFactory=INFO, output
log4j.logger.ee.sk.digidoc.factory.BdocManifestParser=INFO, output
log4j.logger.ee.sk.digidoc.factory.Pkcs12SignatureFactory=INFO, output
log4j.logger.ee.sk.digidoc.factory.BouncyCastleNotaryFactory=INFO, output
log4j.logger.ee.sk.digidoc.tsl.DigiDocTrustServiceFactory=INFO, output
```

```
log4j.logger.ee.sk.digidoc.factory.BouncyCastleTimestampFactory=INFO, output
log4j.logger.ee.sk.xmlenc.factory.EncryptedDataSAXParser=INFO, output
log4j.logger.ee.sk.xmlenc.factory.EncryptedStreamSAXParser=INFO, output
log4j.logger.ee.sk.utils.ConvertUtils=INFO, output
log4j.logger.ee.sk.digidoc.DataFile=INFO, output
log4j.logger.ee.sk.digidoc.SignedDoc=INFO, output
log4j.logger.ee.sk.digidoc.Reference=INFO, output
log4j.logger.ee.sk.xmlenc.EncryptedKey=INFO, output
log4j.logger.ee.sk.digidoc.Base64Util=INFO, output
log4j.logger.ee.sk.digidoc.tsl.TslParser=INFO, output
log4j.logger.ee.sk.digidoc.factory.DigiDocGenFactory=INFO, output
log4j.logger.ee.sk.digidoc.factory.DigiDocServiceFactory=INFO, output
log4j.logger.ee.sk.digidoc.c14n.TinyXMLCanonicalizerHandler_TextStringNormali
zer=INFO, output

#setup output appender
log4j.appender.output =org.apache.log4j.ConsoleAppender
log4j.appender.output.layout=org.apache.log4j.PatternLayout
log4j.appender.output.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss}
[%c{1},%p] %M; %m%n

#setup logfile appender
log4j.appender.logfile=org.apache.log4j.RollingFileAppender
log4j.appender.logfile.File=jdigidoc.log
log4j.appender.logfile.MaxFileSize=512KB
log4j.appender.logfile.MaxBackupIndex=3
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d{ISO8601} %5p [%t] %c(%L)
%x - %m%n
```

Configuring software token usage

Software tokens (PKCS#12 files) can be used for creating technical signatures and decrypting files.

For using software tokens for decryption, set parameter values in JDigiDoc.cfg configuration file as follows:

```
DIGIDOC_SIGN_IMPL = ee.sk.digidoc.factory.Pkcs12SignatureFactory
# DIGIDOC_SIGN_IMPL = ee.sk.digidoc.factory.PKCS11SignatureFactory

DIGIDOC_KEYSTORE_FILE = <your-PKCS#12-keystore-file>
DIGIDOC_KEYSTORE_TYPE = PKCS12
DIGIDOC_KEYSTORE_PASSWD = <your-keystore-password>
```

For digital signing, there are two configuration possibilities:

1. In order to sign with a software token as described in JDigiDoc utility program's command in section "Sample commands of creating technical signatures", sample no 1, add the following parameters to the configuration settings shown above.

```
KEY_USAGE_CHECK = FALSE
DIGIDOC_SIGNATURE_SLOT = 0
```

2. In order to create signature as described in JDigiDoc utility program's command in section "Sample commands of creating technical signatures", sample no 2, only the following parameters need to be configured:

```
DIGIDOC_SIGN_IMPL_PKCS12 = ee.sk.digidoc.factory.Pkcs12SignatureFactory
# DIGIDOC_SIGN_IMPL = ee.sk.digidoc.factory.PKCS11SignatureFactory
KEY_USAGE_CHECK = FALSE
```

Note: signatures that are created with software tokens are technical signatures. Verification of the signatures is expected to produce specific error messages (see also Appendix 2: Signature types).

5. Using JDigiDoc API

5.1 Digital signing

JDigiDoc library offers creating, signing and verification of digitally signed documents, according to XAdES (ETSI TS 101 903 [4]) and XML-DSIG [3] standards. In the next chapters a short introduction is given on the main API calls used to accomplish the above mentioned.

For additional information about the classes and methods described in the following paragraphs, see the full API description (javadoc) that is included in the JDigiDoc library's distribution package.

5.1.1 Initialization

Before you can use JDigiDoc, you must initialize it by reading in configuration data. This is necessary because the library needs to know the location of CA certificates and other parameters in order to fulfill your requests. Pass the full path and name of the configuration file to library like that:

```
ConfigManager.init("jar://JDigiDoc.cfg");
```

The configuration file can be embedded in JDigiDoc jar file which is indicated by the "jar://" prefix. Otherwise just pass the normal full filename in your platform like "/etc/jdigidoc.cfg".

See also "4.1 Loading configuration settings" for alternative configuration loading options.

5.1.2 Creating a DigiDoc document

Before you can add payload data to digidoc document you must create a SignedDoc object to receive this data:

```
SignedDoc sdoc = new SignedDoc(format, version);
//supported format and version combinations: DIGIDOC-XML 1.3 and BDOC 2.1
```

The new digidoc object is kept in memory and not immediately written to a file.

Note: the functionality of creating new files in older DigiDoc file formats SK-XML, DIGIDOC-XML 1.1, DIGIDOC-XML 1.2 and BDOC 1.0 is no longer supported.

Next, set the profile of the document. "TM" profile (using time-marks) is currently the only supported and tested profile in JDigiDoc. Call out the setProfile() method with the following input parameter:

```
sdoc.setProfile(SignedDoc.BDOC_PROFILE_TM);
```

The profile value that is set for SignedDoc object is later (within the same session) used by default as a profile for all signatures that are added to the SignedDoc. Available profiles are described in "3.3 Supported functional properties".

5.1.3 Adding data files

In order to add a data file to a container, the container has to be unsigned and there shouldn't be an existing data file with the same name in the container. Note that only the data file name without path is saved in the document ('/' and '\' characters are not allowed in the data file's name).

Each data file in the container is assigned with an ID value by the library. In case of DIGIDOC-XML 1.3 documents, the ID values in the form of "D<seq_no>" (counting from zero) are used – the respective value is also stored in <DataFile> element's ID attribute. In case of BDOC 2.1

documents, the data file's name is used as the ID value, multiple data files with the same name are not allowed in the container.

NB! In case of BDOC 2.1 documents, it is important to pay attention that the data file's mime type value in manifest.xml file and in signatures*.xml file's <DataObjectFormat><MimeType> element are the same, otherwise the signature is invalid! In case of the ordinary signature creation process, the library sets the correct value automatically. However, if you create a BDOC 2.1 container with JDigiDoc library and want to add signatures*.xml file that you have received from another source (e.g. from DigiDocService) then make sure that the data files' mime-type values in manifest-xml file and in signatures*.xml file are the same.

5.1.3.1 Adding data file from file system

In case of **DIGIDOC-XML 1.3** documents, the data files can be added to DigiDoc container by embedding the data in base64 encoding (using detached data files or embedding pure text or XML is not supported).

```
DataFile df = sdoc.addDataFile(new File((<full-filename-with-path>),
<mime-type>, DataFile.CONTENT_EMBEDDED_BASE64);
```

The new objects are created in memory but the reference files are not read from the disk yet. The library reads all files when you write the digidoc document to a file or start adding signatures (because one has to know the hash codes for signing the data). The data is converted to base64 format.

In case of **BDOC 2.1** documents, the data files are embedded in the container in binary format. If there is no existing manifest.xml file's entry for the data file then this too is created. For adding files to BDOC 2.1 container, do as follows:

```
 DataFile df = sdoc.addDataFile(new File((<full-filename-with-path>),
<mime-type>, DataFile.CONTENT_BINARY);
```

5.1.3.2 Adding data file from byte array in memory

If you don't want JDigiDoc to read the data files from disk because you hold the data in a database, generate it dynamically etc., then assign data to the DataFile object from an in-memory byte array. Once you have assigned the data to this object it will no longer be read from disk.

1. **DataFile.setBody(byte[] data)** method for DDOC and BDOC containers.

Note: In case of DIGIDOC-XML 1.3 files, content of the byte array is automatically converted to base64 by the library, so the input data must not be in base64 format, otherwise double base64 encoding will be done. If you would like to add data which is already in base64 then use method DataFile.setBodyAsData(byte[] data, boolean b64, long len) which is described under the next point.

For example, in case of DIGIDOC-XML 1.3 documents, do as follows:

```
DataFile dFile = new DataFile(sdoc.getNewDataFileId(), //gets the next
      // datafile index as ID value
      DataFile.CONTENT_EMBEDDED_BASE64, // content type for DDOC files
      "test.txt", // data file name
      "text/plain", // mime type
      sdoc); // SignedDoc object created earlier

String myBody = "My sample data"; //e.g. data read from
      // database
byte bodyArray[] = myBody.getBytes();
df.setBody(bodyArray); // the data content is added
```

In case of BDOC 2.1 documents (supported since v3.8.1):

```
DataFile dFile = new DataFile("test.txt", //set the data file name as
      // ID value in case of BDOC documents
      DataFile.CONTENT_BINARY, // content type for BDOC files
      "test.txt", // data file name
      "text/plain", // mime type
      sdoc); // SignedDoc object created earlier

String myBody = "My sample data"; //e.g. data read from
      // database
byte bodyArray[] = myBody.getBytes();
df.setBody(bodyArray); // the data content is added
```

2. **DataFile.setBodyAsData(byte[] data, boolean b64, long len)** method for DDOC files (since v3.8.1) – if the boolean parameter b64 is set to true then the original data is not converted to base64.
3. **DataFile.setBase64Body(byte[] data)** method for DDOC files - has the same functionality as DataFile.setBody(byte[] data). The input data is always converted to base64 format.

5.1.4 Adding signatures

It is possible to add a signature to a container only if it contains at least one data file, multiple signatures can be added to a single container. The signer's certificate and PIN code to access the private signature key are required during signing.

The SignatureFactory interface is used for signing. You can sign either by:

- using an Estonian ID card or
- any other smartcard or token provided that you have the external native language PKCS#11 driver for it;
- using a HSM device if you have the external native language PKCS#11 driver for it;
- using a software token (PKCS#12 file);
- calculate the signature in some external program (e.g. web-application) and then add the signature value to digidoc document.

Note: the functionality of adding signatures and removing existing signatures is restricted in the following cases:

- in case of older DigiDoc file formats SK-XML, DIGIDOC-XML 1.1, DIGIDOC-XML 1.2 and BDOC 1.0.
- in case of BDOC 2.1 files where SHA-1 hash function has been used.

Before signing you have to get the signer's certificate that is being referenced by the signature.

If you use **PKCS#11 driver** to access smart card then do:

```
String pin = "<smartcard-PIN>";
```

Initialize the SignatureFactory according to configuration file's parameters.

```
SignatureFactory sigFac = ConfigManager.instance().getSignatureFactory();
```

Get the signing certificate and log in with PIN code:

```
X509Certificate cert = sigFac.getCertificate(0, pin);
```

The first parameter of the above mentioned method is the slot number (index of the key pair used for signing). Note that the slot number denotes the sequence number (counting from zero) of the signature certificate among all signature certificates on the identity token (i.e. the value may not correspond to the actual slot ID value in PKCS#11 driver). In this sample we

use 0 as it's used on Estonian ID cards. Please note that when using the default KEY_USAGE_CHECK configuration setting then this index starts with 0 and counts ONLY the key pairs usable for digital signature, e.g. no authentication key pairs!

Now compute the data file's hash codes and create a Signature object:

```
Signature sig = sdoc.prepareSignature(cert,
      null, // String[] roles - optional, role / resolution of the signer
      null); //SignatureProductionPlace, optional, signer's address
```

When adding a role or resolution of the signer then at most one entry is allowed in the roles array (the second parameter of the prepareSignature() method shown above). The entry can contain only the signer's role or role along with the signer's resolution, separated with a slash character, i.e. **"role / resolution"**. The value will be written to a single <ClaimedRole> xml element in the file. When adding the signer's resolution then role must also be added. Note that during signature validation, at most two <ClaimedRole elements are allowed due to historical reasons.

Note: if you have read in an unsigned BDOC container from an input steam and are adding a signature to it then it is necessary to additionally determine the signature profile before signing (even if the profile has previously been set for the container). In this case, set the profile as follows:

```
sig.setProfile(SignedDoc.BDOC_PROFILE_TM);
```

The signature is not complete yet as it's missing the actual RSA signature data. We have to now calculate the final hash value (SHA-1 in case of DIGIDOC-XML 1.3 and by default, SHA-256 in case of BDOC 2.1) that serves as input for RSA signature:

```
sidigest = sig.calculateSignedInfoDigest();
```

In case of PKCS#11 driver, compute RSA signature value as follows (alternatively, you can use a web browser plugin to get the RSA signature value):

```
byte[] sigval = sigFac.sign(sigdigest, 0, pin, sig);
```

The second parameter specifies the slot number, as described in method sigFac.getCertificate(int slot, String pin); above.

Finally, add signature value to the Signature object:

```
sig.setSignatureValue(sigval);
```

Alternatively, if you use HSM device for signing then replace the signature value calculation in the example above with the following method. Note that support for HSM device in JDigiDoc is experimental.

```
byte[] PKCS11SignatureFactory.sign(byte[] sidigest, //signed info digest
      long nSlotId, // certificate slot's ID value
      String certLabel, // label name of the certificate object
      String pin, // pin code for accessing the slot
      Signature sig); // Signature object
```

Signature certificate on HSM device is determined by its slot ID number and the certificate object's label, both of the parameters are mandatory. Note that the slot ID used in the current method refers to the actual ID value of the slot (not the sequence number of the certificate on device, as used in other JDigiDoc methods). Also, the signature certificate and private signature key have to be in the same slot and must have same label values (i.e. the label values of the certificate and private key objects are used to match the certificate with the appropriate private key).

5.1.5 Adding an OCSP confirmation

Call the following method to add OCSP confirmation (time-mark):

```
sig.getConfirmation();
```

After adding an OCSP confirmation, the signature is now complete and provides long-time proof of the signed data. Note that at most one OCSP confirmation is allowed for a signature.

OCSP confirmation's data are stored in element <UnsignedSignatureProperties> in the DigiDoc file (see also [1], section "XAdES extension block – unsigned parameters").

5.1.6 Reading and writing digidoc documents

Write a SignedDoc object to a digidoc file as follows:

```
sdoc.writeToFile(new File("<full-path-and-filename>"));
```

If you want to store the digidoc document in database not in a file, then use the method:

```
SignedDoc.writeToStream(OutptStream os);
```

To read a DigiDoc document, you can use one of the following methods:

- DigiDocFactory.readSignedDoc(String fileName)
- DigiDocFactory.readSignedDocOfType(String fname, boolean isBdoc, List lerr)
- SAXDigiDocFactory.readSignedDocFromStreamOfType(InputStream is, boolean isBdoc, List lerr)

For example:

```
DigiDocFactory digFac = ConfigManager.instance().getDigiDocFactory();
SignedDoc sdoc = digFac.readSignedDoc("<full-path-and-filename>");
```

5.2 Validating signed documents

Validation of a signed DigiDoc document consists of three main steps:

1. Read in the DigiDoc document and check for returned parsing errors;
2. Call out the main validation method of the library;
    a. Check for fatal errors. End the process with INVALID status if any fatal errors have been discovered.
3. Check for additional errors/warnings (separate method calls);
4. Determine the validation status of the document (according to the returned error codes and validation status priorities).

Note: steps 1a, 2 and 3 are additions to the validation process since the library's version v3.8.

5.2.1 Reading and parsing the DigiDoc document

Read in and parse the DigiDoc document. Errors that are discovered during parsing can be collected to a List by using either of the following methods:

```
// Reading from disk:
SAXDigiDocFactory.readSignedDocOfType(String fname,
      boolean isBdoc,
      List lerr); // list for collecting parsing errors
```

```
// Reading from an input stream:
SAXDigiDocFactory.readSignedDocFromStreamOfType(InputStream is,
      boolean isBdoc,
      List lerr); // list for collecting parsing errors
```

After reading in the document, continue with using the main validation method of the library. If the returned List container contains any exceptions then keep them for determining the validation status later on.

5.2.2 Using the main validation method

Secondly, validate the whole DigiDoc document as follows (e.g. after having read in a digidoc document as described in the previous chapter):

```
ArrayList<Exception> errs = SignedDoc.verify(true,
      true); // The method's argument values are obsolete and don't have
      //any actual meaning.
```

This method validates the document's structure and if there are no fatal errors, validates all signatures one by one. If the signature has an OCSP confirmation then this too is being validated.

If document structure's validation process discovers fatal errors in the XML structure then validation is cancelled and the respective exception is returned to the user. If document's structure is validated with no fatal errors then the method continues with validating signatures and OCSP confirmations (OCSP responses added to the document). In case of signature validation errors, no exceptions are actually thrown, but they are returned to the user in an ArrayList container. This way you can get all errors and not just the first.

After calling out the main validation method:

- If the returned ArrayList container contains any exceptions then it must be checked if any of the exceptions were fatal (e.g. the document's XML structure is broken and it is not possible to continue the validation process). You can check for fatal errors with the following method:

  ```
  SignedDoc.hasFatalErrs(ArrayList lerrs); //returns TRUE if fatal errors
        // are found
  ```

  If the method returns "true" then you must terminate the validation process with status INVALID. Otherwise, continue with checking for additional errors/warnings described in the next chapter.

- If the returned ArrayList container is empty, continue with checking for additional errors/warnings described in the next chapter.

5.2.3 Checking for additional errors/warnings

There are validation cases that are not checked in the default validation method of the library, instead, separate methods for checking the specific situations have to be called out by the library's user. In JDigiDoc library, checking for a **test signature** must be done separately.

The following subchapters describe how these checks can be implemented. After checking for additional errors/warnings, collect all of the error codes and continue with determining the validation status as described in the next chapter.

5.2.3.1 Checking for test signature

Test signature is a signature that has been created by using test certificates (e.g. signer's certificate and/or OCSP responder server's certificate have been issued for testing purposes).

1. For identifying if a certificate is a SK issued test certificate, use the following method with the signer's certificate as input:

```
ee.sk.digidoc.factory.DigiDocGenFactory.isTestCard(X509Certificate cert);
```

   The identification is done with comparing certificate policy OID values.

2. It must also be checked separately, if the OCSP responder was issued by SK's test service:

```
ee.sk.digidoc.factory.DigiDocVerifyFactory.verifySignatureFromLiveAndOcspFr
omTest(Signature sig, List lerrs)
```

5.2.4 Determining the validation status

After validating the signed DigiDoc document, the validation result must be determined by the library's user. Final validation result must be one of the possible validation statuses that are described in the table below, the status must be chosen according to its priority.

The validation status priorities have to be applied in two cases:

1. **Returning a validation result of a single signature:**
   If there are more than one validation errors that occur when validating a single signature in DigiDoc container then the overall status of the signature should be chosen according to the status priorities.

2. **Returning a validation result of the whole DigiDoc container:**
   If there are more than one signatures in a DigiDoc container and the signatures have different validation statuses or validation of the container structure returns a different status then the overall status of the DigiDoc file should be chosen according to the status priorities.

NB! User of the library has to determine the validation status according to the error code that is returned by the library's validation method.

Priority	Status	Error code	Description
1	**INDETERMINATE/UNKNOWN**	**92** ERR_CERT_UNKNOWN **39** ERR_SIGNERS_CERT	Validation process determines that one or more of the certificates included in the document are unknown or not trusted, i.e. the certificates have been issued by an unknown Certificate Authority (the CA has not been added to trusted list). **Notes**: • The file and signature(s) are not legally valid. • If the CA will later be added to the trusted list/trust store then the validation status can change to any of the other statuses described in the current table. **Suggested warning message** (also displayed in DigiDoc3 Client): "Signature status is displayed as unknown if you don't have all validity confirmation service certificates and/or certificate authority certificates installed into your computer" **More info**: http://www.id.ee/index.php?id=35941 **Sample file**: unknown_CA.asice

Priority	Status	Error code	Description
2	**INVALID**	All errors except of the ones that are regarded as warnings by the library's user.	Validation process returns error(s), the errors have not been explicitly determined as minor error(s) by the library's user. **Note:** • The file and signature(s) are not legally valid. • No further alterations should be made to the file, i.e. no signatures should be added or removed.
3	**TEST**	**178** ERR_TEST_SIGNATURE	Test certificates have been used in the signed file (e.g. signer's certificate and/or OCSP responder server's certificate have been issued in testing purposes). **Notes**: • Test signature is not legally binding even if the signature is valid. • This status is used in combination with the other validation statuses described in the current table. **Suggested warning message** (also displayed in DigiDoc3 Client): "Test signature" **More info**: http://www.id.ee/index.php?id=30494 **Sample file**: test_signature.ddoc
4	**VALID WITH WARNINGS**	See the next section.	Validation process returns error(s) that have been previously explicitly categorized (by the library's user) as minor technical errors. Note that this status is used only in exceptional cases, more details of which are given in the next chapter. **Notes**: • The file and signature(s) are handled as legally valid. • The error(s) are regarded as validation warnings. • Validation warnings should be displayed to the user. • No further alterations should be made to the file, i.e. no signatures should be added or removed. • Creator of the file should be informed about the error situation.
5	**VALID**	N/A	Validation process returns no errors. The signature is legally valid.

The error codes described in the table above are defined in ee.sk.digidoc.DigiDocException class.

Sample code of DigiDoc file validation can be found from ee.sk.test.jdigidoc class, from the following method:

```
runValidateSignedDocCmds(String[] args); //utility program's command -ddoc-validate
```

5.2.4.1 Validation status VALID WITH WARNINGS

In special cases, validation errors can be regarded as minor technical errors and the file's validation status can be regarded as VALID WITH WARNINGS instead.

NB! User of the DigiDoc library has to decide on his/her own when to use VALID WITH WARNINGS status instead of INVALID: there may be different interpretations of the severity of validation errors in different information systems then the final decision when to use this status has to be made by the library's user according to the requirements of the specific information system.

It is recommended to use the validation status VALID WITH WARNINGS in case of the error situations that are included in the table below - these error situations are regarded as VALID WITH WARNINGS in DigiDoc applications and software libraries, including:

- DigiDoc3 Client desktop application,
- JDigiDoc, Libdigidocpp and CDigiDoc software libraries' utility programs.

Table 1. Validation error codes recommended to be handled as VALID WITH WARNINGS

Status	Error code	Related DigiDoc file format	Description
VALID WITH WARNINGS	**129** WARN_WEAK_DIGEST	BDOC 2.1	Weaker digest method (SHA-1) has been used than recommended when calculating either <Reference> or <Signature> element's digest value. **Suggested warning message** (also displayed in DigiDoc3 Client): "The current BDOC container uses weaker encryption method than officially accepted in Estonia." **Sample file**: weak-warning-sha1.bdoc
	173 ERR_DF_INV_HASH_GOOD_ALT_HASH	DDOC 1.0 DDOC 1.1 DDOC 1.2 DDOC 1.3	<DataFile> element's xmlns attribute is missing. **Suggested warning message** (also displayed in DigiDoc3 Client): "This DigiDoc documents has not been created according to specification, but the digital signatures is legally valid. You are not allowed to add or remove signatures to this container." **More info**: http://www.id.ee/?id=36213 **Sample file**: datafile_xmlns_missing.ddoc
	176 ERR_ISSUER_XMLNS	DDOC 1.1 DDOC 1.2 DDOC 1.3	<IssuerSerial><X509IssuerName> and/or <IssuerSerial><X509SerialNumber> element's xmlns attribute is missing. **Suggested warning message** (also displayed in DigiDoc3 Client): "This DigiDoc documents has not been created according to specification, but the digital signatures is legally valid. You are not allowed to add or remove signatures to this container." **More info**: http://www.id.ee/?id=36213 **Sample file**: issuerserial_xmlns_missing.ddoc

Status	Error code	Related DigiDoc file format	Description
	177 ERR_OLD_VER	DDOC 1.0 DDOC 1.1 DDOC 1.2 BDOC 1.0	DigiDoc file's version is older than currently supported. Note that the error situation affects only the container and not the signatures, therefore, in DigiDoc libraries, it is returned and displayed only at container level. **Suggested warning message** (also displayed in DigiDoc3 Client): "The current file is a DigiDoc container that is not supported officially any longer. You are not allowed to add or remove signatures to this container" **More info**: http://www.id.ee/index.php?id=36161 (DDOC) http://www.id.ee/?id=36161 (BDOC 1.0) **Sample file**: old_digidoc_format_1.0.ddoc **Sample file**: old_bdoc_format_1.0.bdoc

Sample code for determining validation warnings can be found from jdigidoc.java utility program (ee.sk.test.jdigidoc). See command –ddoc-validate (ee.sk.test.jdigidoc.runValidateSignedDocCmds(String[] args)) and method ee.sk.test.jdigidoc.isWarning().

5.2.5 Additional information about validation

5.2.5.1 Validating document's structure separately

For validating the XML structure of a digidoc document, JDigiDoc classes contain methods that validate the contents of XML fields and attributes.

It is always useful to validate a digidoc document after reading it from a file or after adding or changing some content. This helps to identify problems at later phases. Note that this method is also called out by the library's main validation method SignedDoc.verify(boolean, boolean).

XML structure can be validated with the method:

```
ArrayList SignedDoc.validate(true);
```

This method returns an array of DigiDocException objects. If the array is empty then the document's structure is ok.

5.2.5.1 Validating signatures separately

It is possible to validate only one signature object separately from the rest of the document with a single method call (since the library's version v3.8). It can be useful to distinguish a single signature's validation errors from the rest of the DigiDoc document's validation result.

Use the following method to validate the Signature object's XML structure and cryptographic value:

```
ArrayList Signature.verify(SignedDoc sdoc, ArrayList lerrs);
```

5.2.5.2 Overview of validation activities

Overview of validation activities is as follows:

1. checking that all the data files and signature's meta-data (signer's role, etc.) are included in the signature by calculating the data objects' digest values and comparing them with the <Reference> element values in the signature;
2. checking that the claimed signer's certificate is the actual certificate that was used for signing; checking that the "Non-repudiaton" value is set in the "Key Usage" extension of the signer's certificate;
3. checking that the signature value is correct by decrypting the value with the signer's public key and comparing the result with digest calculated from <SignedInfo> element block;
4. checking that the OCSP response confirms the signer certificate's validity and corresponds to the signature value (by comparing the digest value of <SignatureValue> element's value and OCSP response's nonce value);
5. checking that the signer's and OCSP responder's certificates are trusted (i.e. the certificates' issuers are registered in trust store, i.e. the configuration file).

Note that verifying a DDOC signature that has no OCSP confirmation produces an error message "Error: 128 - Signature has no OCSP confirmation!". If the signature that is being verified was created with a software token (PKCS#12 file) then error message "Error 39: Signer's cert does not have non-repudiation bit-set!" is produced.

5.3 Encryption and decryption

5.3.1 Format of the encrypted file

In addition to digital signing, JDigiDoc library offers also digital encryption and decryption according to the ENCDOC-XML 1.0 (based on XML-ENC) standard. This standard describes encrypting and decrypting XML documents or parts of them and it also allows encrypting any binary data in Base64 encoding.

The ENCDOC-XML 1.0 encrypted file format (also referred to as CDOC 1.0) has been described in [12]. The format uses .cdoc as encrypted file extension.

Data is encrypted with a 128 bit AES transport key which is in turn encrypted with the recipient's certificate. Encryption scheme is therefore certificate-based – it is possible to encrypt data using public key component fetched from some certificate. The decryption can be performed only by using private key corresponding to that certificate.

Many encrypted data objects or a mix of encrypted and unencrypted data in one XML document is not supported.

One encrypted document:

- contains only one <EncryptedData> element, which is also the document's root element
- contains one <EncryptedKey> element for every recipient (i.e. possible decrypter) of the document
- contains a set of <EncryptionProperty> elements to store any meta data.

If it is needed to incorporate multiple data files into one encrypted document then it is possible to pack the files to a single container and then encrypt the container.

5.3.2 Encryption

In the following chapters we review most common encryption operations with JDigiDoc library.

The process to compose an encrypted document is as follows:

1. create the EncryptedData object first,
2. add all recipients' certificates and other data,
3. add encryption properties,
4. add the unencrypted data and encrypt it,
5. finally store it in a file or other medium.

5.3.2.1 *Creating EncryptedData object*

```
EncryptedData cdoc = new EncryptedData(
      null, // optional Id attribute value
      null, // optional Type attribute value
      null, // optional MimeType attribute value
      EncryptedData.DENC_XMLNS_XMLENC,  // fixed xml namespace
      EncryptedData.DENC_ENC_METHOD_AES128); // fixed cryptographic
algorithm
```

Optional attribute values have to be passed in as nulls in case you don't need them. Passing in for example an empty string will case this to be considered a valid attribute value. More information about the attribute values can be found from [12].

The "MimeType" attribute of <EncryptedData> element reflects the mime type of the original data that is encrypted. The default value „application/octet-stream" can be used, except in the following cases (for compatibility with DigiDoc3 Crypto application):

1. If encrypting a BDOC 2.1 document, then the "MimeType" attribute should be set as:

   `"application/vnd.etsi.asic-e+zip"`

   which has also been defined as constant: `SignedDoc.MIMET_FILE_CONTENT_20`

2. If encrypting a DIGIDOC-XML 1.3 document (and also when using a temporary DDOC container for encapsulating the original files to be encrypted[2]), you should assign the "MimeType" attribute following value:

   `"http://www.sk.ee/DigiDoc/v1.3.0/digidoc.xsd"`

   which has also been defined as a constant: `EncryptedData.DENC_ENCDATA_TYPE_DDOC`

Note: the attribute "MimeType" was also historically used to store the fact that the data has been packed with ZLIB algorithm before encryption. As compressing data during encryption is deprecated since v3.9 of the library then this case should not be used any longer. If compression has been used nevertheless, then the library automatically assigns the following value to "MimeType" attribute:

`"http://www.isi.edu/in-noes/iana/assignments/media-types/application/zip"`

If the "MimeType" attribute was already set by the library's user then the original "MimeType" value is stored in an <EncryptionProperty Name="OriginalMimeType"> subelement. If

[2] When using temporary intermediary DDOC container then the data file(s) to be encrypted are placed inside an unsigned DDOC container (DIGIDOC-XML 1.3). The whole DDOC container is then encrypted. The unsigned DDOC container should be discarded when decryption is done, e.g. as it is done by DigiDoc3 Client, so that the user gets original data file(s) as a result.

NOTE: it is recommended to use direct encryption without temporary DDOC containers as much as possible. Direct encryption/decryption is supported in all of the DigiDoc libraries with CDOC support and in DigiDoc3 Crypto versions 3.9 and above.

JDigiDoc reads a document with this specific "MimeType" then it decompresses the decrypted data and restores the original mime type if one is found.

5.3.2.2 Adding recipient info

Every encrypted document should have at least one recipient block, otherwise nobody can decrypt it.

It is possible to encrypt for multiple certificates at once. Certificates for encryption are fetched from a file in the file system (DER and PEM encoding are supported), possible sources for finding them can be:

- Windows Certificate Store ("Other Persons")
- LDAP directories (for Estonian ID card holders, all valid certificates are available at: ldap://ldap.sk.ee, see also https://www.sk.ee/en/repository/ldap/).
- ID-card in smart-card reader.

Note that in JDigiDoc library, the certificates that can be used for encryption must have the value "Key Encipherment" included in "Key Usage" attribute field. In case of Estonian ID cards and Digi-ID it's the authentication certificate. The recipient's certificate must be in PEM format.

NB! Encryption should be done for the authentication certificates on all the recipient's valid identity tokens, i.e. if the recipient has a valid ID-card and Digi-ID card then encryption should be done for the certificates on both of the tokens. Mobile-ID authentication certificate should not be used for encryption as decryption with Mobile-ID is not possible.

For example, the following process can be used to select the appropriate certificate(s) to be used for encryption:

1. Check that the certificate's "KeyUsage" field includes "KeyEncipherment" value
2. Check that the certificate is not a Mobile-ID certificate by inspecting the certificate policy value in "CertificatePolicies" field. In case of a Mobile-ID certificate, the certificate policy identifier value is a string starting with "1.3.6.1.4.1.10015.1.3." or "1.3.6.1.4.1.10015.11.1.".

For every recipient the library creates a <EncryptedKey> element and stores the following data within that element:

- the AES transport key encrypted with the recipients certificate
- the recipient's certificate itself
- possibly some other data used to identify the key.

EncryptedKey object with the recipient's data can be added as follows:

```
X509Certificate recvCert = SignedDoc.readCertificate(new File(certFile));
EncryptedKey ekey = new EncryptedKey(
      null,  // optional Id attribute value
      null,  // optional Recipient attribute value
      EncryptedData.DENC_ENC_METHOD_RSA1_5, // fixed cryptoalgorithm
      null,  // optional KeyName subelement value
      null,  // optional CarriedKeyName subelement value
      recvCert); // recipients certificate. Required!
cdoc.addEncryptedKey(ekey);
```

Optional attributes "Id", "Recipient" and/or subelements <KeyName> and <CarriedKeyName> can be added to identify the key object. All of the above mentioned attributes and subelements are optional but can be used to search for the right recipient's key or display its data in an application.

The command line utility program jdigidocutil-*.jar assigns a unique value to every EncryptedKey objects "Recipient" attribute. It could be the recipients forename or something more complicated like "<last-name>,<first-name>,<personal-code>". This can later be used as a command line option to identify the recipient whose key and smart card is used to decrypt the data.

As the recipient's certificate is the only required data, it would be wise not to demand encrypted documents to contain other attributes for an application's proper functioning. Something from the certificate like its CN attribute should be used to identify the recipient.

5.3.2.3 Setting the encryption properties

The encrypted CDOC document can contain a number of <EncryptionProperty> elements that can be used to store various meta-data. The JDigiDoc library automatically sets the "LibraryVersion" and "DocumentFormat" properties, other properties are optional.

If you are using temporary intermediary DDOC container for encapsulating the original files to be encrypted then the original files need to be placed inside an unsigned DDOC container (DIGIDOC-XML 1.3) and then encrypted (and the Mimetype value set accordingly, see chap. 5.3.2.1). In this case, the encryption property "orig_file" must be specified.

Note: it is recommended to use direct encryption without temporary DDOC containers as much as possible. Direct encryption/decryption is supported in all of the DigiDoc libraries with CDOC support and in DigiDoc3 Crypto versions 3.9 and above.

The "orig_file" properties must be defined as follows. There must be one "orig_file" property for every data file (<DataFile> element) in the DDOC container, each property must be in the following format:

`<file-name>|<size-in-bytes>|<mime-type>|<DataFile-ID-in-DDOC>`

For example:

```
// Create DDOC intermediate file
SignedDoc sdoc = new SignedDoc(SignedDoc.FORMAT_DIGIDOC_XML,SignedDoc.VERSION_1_3);

// Add the unencrypted data file to DDOC container
File fIn = new File(inFile);
byte[] data = SignedDoc.readFile(fIn);
DataFile df = sdoc.addDataFile(fIn, "<mime-type>",
       DataFile.CONTENT_EMBEDDED_BASE64);
df.setBase64Body(data);

// Add the DDOC data to CDOC container
byte[] inData = sdoc.toXML().getBytes("UTF-8");
cdoc.setData(inData);
cdoc.setDataStatus(EncryptedData.DENC_DATA_STATUS_UNENCRYPTED_AND_NOT_COMPRESSED);

cdoc.setMimeType(EncryptedData.DENC_ENCDATA_TYPE_DDOC); // ensure that the MimeType
       // attribute is set accordingly

// Create orig_file property
StringBuffer sb = new StringBuffer();
sb.append(fIn.getName());
sb.append("|");
sb.append(new Long(fIn.length()).toString());
sb.append("|");
sb.append(df.getMimeType());
sb.append("|");
sb.append(df.getId());
cdoc.addProperty(EncryptedData.ENCPROP_ORIG_FILE, sb.toString()); // add to cdoc

// Continue with encryption...
```

5.3.2.4 Encryption and data storage

There are two possible methods for encrypting data, depending on the size of the data object that is being encrypted. In both cases it isn't necessary to use files to store encrypted data. It can be can written to any output stream and used as required.

Note: since v3.9 of the library, functionality of compressing the data during encryption is deprecated. By default, the library never compresses data during encryption (by using the EncryptedData.DENC_COMPRESS_NEVER option). Historically, the EncryptedData.DENC_COMPRESS_ALLWAYS option was used to always compress the input data and EncryptedData.DENC_COMPRESS_BEST_EFFORT option enabled to use compression only if it resulted in reduction of the input data size.

Encrypting small data objects

In case of small data objects, it is possible to do all operations in memory. The method is faster and more flexible but requires more memory.

For example:

```
// Read unencrypted data
byte[] inData = SignedDoc.readFile(new File(inFile));
cdoc.setData(inData);
cdoc.setDataStatus(EncryptedData.DENC_DATA_STATUS_UNENCRYPTED_AND_NOT_COMPRESSED);

// Encryption
cdoc.encrypt(EncryptedData.DENC_COMPRESS_NEVER);

FileOutputStream fos = new FileOutputStream(outFile);
fos.write(m_cdoc.toXML());
fos.close();
```

Encrypting big data objects

Encryption of big data objects is done by reading and handling all data in blocks of fixed size. The method is capable of encrypting large sets of data but less flexible. It doesn't offer the option of encrypting in memory, so input and output streams must be provided. Note that the functionality of encrypting big data sets is not currently tested.

```
cdoc.encryptStream(new FileInputStream(inFile),
       new FileOutputStream(outFile), EncryptedData.DENC_COMPRESS_NEVER);
```

5.3.3 Parsing and decryption

Note: when decrypting files then it should be taken into account sometimes the data file that has been encrypted is placed inside **a temporary DDOC container before encryption**. In this case, it is also necessary to extract the original data file(s) from DigiDoc container after decryption. It is possible to detect if a temporary DDOC container has been used in the following way:

- <EncryptedData> element's "Mimetype" attribute value is „http://www.sk.ee/DigiDoc/v1.3.0/digidoc.xsd".
- There are one or more EncryptionProperty Name="orig_file"> elements in the following format: <file-name>|<size-in-bytes>|<mime-type>|<DataFile-ID-in-DDOC>
- After decryption, an unsigned DDOC file is received as a result.
- In other cases, it can be assumed that the data has been encrypted directly, without the temporary DDOC container.

There are two methods available for decrypting and parsing encrypted documents.

1. EncryptedDataParser – suitable for parsing smaller encrypted objects.

   After parsing, data is in memory and can be decrypted or displayed on screen. It does not automatically decrypt data during parsing. Decryption is a separate operation.

   Parsing small encrypted files is done as follows:

```
EncryptedDataParser dencFac = ConfigManager.instance().
      getEncryptedDataParser();
cdoc = dencFac.readEncryptedData(inFile);
```

   Now all data is in memory in encrypted and possibly in compressed form.

   The methods of EncryptedData, EncryptedKey and EncryptionProperty objects can be used to display and decrypt data as follows:

```
cdoc.decrypt(0,  // index of EncryptedKey object
      0,      // smartcards Token index. For Estonian ID cards always 0
      pin);   // smartcards PIN code. For Estonian ID card PIN1
FileOutputStream fos = new FileOutputStream(outFile);
fos.write(cdoc.getData());
fos.close();
```

2. EncryptedStreamParser – suitable for parsing and decrypting large encrypted objects.

   Doesn't keep any data in memory, input and output streams have to be provided. Decryption and decompression is done during parsing. Note that the functionality of decrypting big data sets is not currently tested.

   For decrypting big encrypted documents, you firstly need to set up the input and output streams:

```
// provide input and output streams
FileInputStream fis = new FileInputStream(inFile);
FileOutputStream fos = new FileOutputStream(outFile);
EncryptedStreamParser streamParser = ConfigManager.
      instance().getEncryptedStreamParser();
```

   Next, call one of the following decryption methods. The methods read the data from input stream, decrypt, possibly decompress it and write it to output stream.

   - Method decryptStreamUsingRecipientName() – the EncryptedKey object is identified with the "Recipient" attribute. Only the PKCS#11 token type is supported.

```
streamParser.decryptStreamUsingRecipientName(fis, fos,
      0,  // smartcard's token index. For Estonian ID cards always 0
      pin, // smartcard's PIN code. PIN1 for Estonian ID cards
      recvName); // selected EncryptedKey object's Recipient attribute
```

   - Method decryptStreamUsingTokenType() – allows you to choose the appropriate token type for decryption (PKCS#11 and PKCS#12 tokens are supported).

     In case of PKCS#11, do as follows:

```
streamParser.decryptStreamUsingTokenType(fis, fos,
      0,  // PKCS11 token index. For Estonian ID cards always 0
      pin, // PIN code to decrypt with PKCS11. PIN1 for Estonian ID cards
      SignatureFactory.SIGFAC_TYPE_PKCS11, // token type: PKCS11 or PKCS12
      null); //PKCS12 keystore filename and path if PKCS12 is used.
             // Set the value to null in case of PKCS11
```

- If you use HSM device for decryption then call method:

```
EncryptedStreamParser.decryptStreamUsingRecipientSlotIdAndTokenLabel(
      InputStream dencStream, // input stream
      OutputStream outs, // output stream
      int slot, // slot ID of the decryption certificate
      String label, // label name of the certificate object
      String pin); // pin code to access the certificate's slot
```

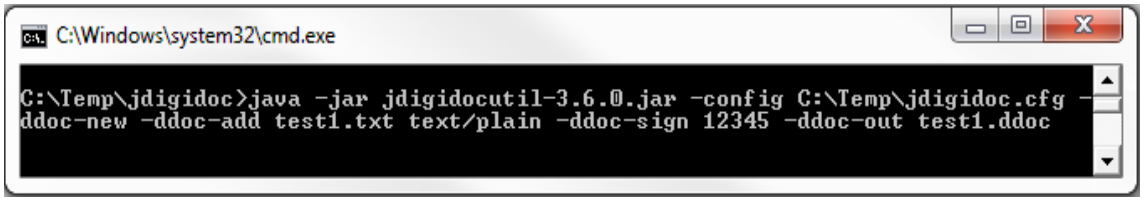
Decryption certificate on HSM device is determined by its slot ID number and the certificate object's label, both of the parameters are mandatory. Note that the slot ID used in the current method refers to the actual ID value of the slot (not the sequence number of the certificate on device, as used in other JDigiDoc methods). Also, the decryption certificate and accompanying private key have to be in the same slot and must have same label values (i.e. the label values of the certificate and private key objects are used to match the certificate with the appropriate private key).

Finally, close the input and output streams:

```
fos.close();
fis.close();
```

6. JDigiDoc utility program

The command line utility program which is included in the JDigiDoc distribution as an executable JAR archive file **jdigidocutil-*.jar** can be used to test the library or simply use it directly to encrypt, decrypt and digitally sign documents ('*' in the file name denotes the version number of the JDigiDoc library).



7 Using the command line utility program with Windows Command Prompt

The general format for executing the program is:

```
> java –jar jdigidocutil-*.jar [commands]
```

A list of all the available commands and their format can always be displayed by using the -? or –help commands:

```
> java –jar jdigidocutil-*.jar -help
```

The jdigidocutil-*.jar JAR archive contains a metadata file **META-INF/MANIFEST.MF** which specifies the necessary meta-information for executing the JDigiDoc utility program. For example, the MANIFEST.MF file specifies the main Java class of the program (ee.sk.test.jdigidoc) and defines all of the necessary classpath variables.

Note: classpath values for using Estonian CA's test certificates and Lithuanian CA's certificates have also been pre-defined in the manifest file – /esteidtestcerts.jar and /lib/esteidtestcerts.jar for Estonian CA's test certificates; /jdcerts.jar and /lib/jdcerts.jar for supported Lithuanian CA's certificates. For more information on using the mentioned certificates, see sections 5.1.2 Trusted Estonian certificate authorities, under "Supported SK test certificate hierarchy chains" and section 5.1.4 Trusted Lithuanian certificate authorities.

6.1 General commands

Note: the utility program commands' parameters that are marked between "<" and ">" characters are mandatory and have to be specified. The parameters between "[" and "]" characters are optional.

- **-? or –help** – displays help about command syntax
- **-config <configuration-file>** - specifies the JDigiDoc configuration file name.
- **-check-cert <certificate-file-in-pem-format>** - checks the certificate validity status

Setting the configuration file

-config <configuration-file>

You can dynamically specify the configuration file used before executing each command line task.

If left unspecified, then the default configuration file is looked up from locations included in the classpath: "jar://jdigidoc.cfg".

Checking the certificate

-check-cert <certificate-file-in-pem-format>

Used for doing a preliminary check of the chosen certificate's validity; returns an OCSP response from the certificate's CA OCSP responder.

Returns the certificate's validity information:

- GOOD – certificate is valid
- REVOKED – certificate has been revoked
- UNKNOWN – certificate has never been issued or issuer is unknown
- EXPIRED – certificate has been expired
- SUSPENDED – certificate has been suspended
- OCSP_UNAUTHORIZED – if no access to OCSP validity confirmation service

Sample: setting the configuration file when creating a new DigiDoc container

```
> java –jar jdigidocutil-*.jar –config c:\temp\jdigidoc.cfg -ddoc-new -ddoc-
add c:\temp\test1.txt text/plain -ddoc-out c:\temp\test1.ddoc

  Input:
  -c:\temp\jdigidoc.cfg – the configuration file to be used
  -c:\temp\test1.txt    - a data file to be added to ddoc container
  - text/plain          - mime type of the data file
  - c:\temp\test1.ddoc- ddoc container to be created
```

6.2 Digital signature commands

Note: the utility program commands' parameters that are marked between "<" and ">" characters are mandatory and have to be specified. The parameters between "[" and "]" characters are optional.

- -**ddoc-in <input-digidoc-file> -** reads in a DigiDoc file
- -**ddoc-in-stream <input-digidoc-file> -** reads in a DigiDoc file from inputstream. Used for testing API's inputstream functions.
- **-ddoc-in-ostream <input-digidoc-file>** - reads in a DigiDoc file from java.io.ObjectInputStream and deserializes it as a SignedDoc object. Note that the command is currently not being tested.
- **-ddoc-new [format] [version] –** creates a new DigiDoc container
- **-ddoc-add <input-file> <mime-type> [content-type] –** adds a data file to a DigiDoc container
- **-ddoc-add-mem <input-file> <mime-type> [content-type] –** adds a data file to a DigiDoc container by internally using an in-memory byte array for keeping the data file's content.
- **-ddoc-sign <pin-code> [role/resolution] [country] [state] [city] [zip] [slot] [profile] [driver] [keystoreFile] –** signs a DigiDoc file

- **-ddoc-out <output-file> -** creates a DigiDoc file at the specified location
- **-ddoc-out-stream <output-file> -** writes the DigiDoc file to an outputstream. Used for testing API's outputstream functions.
- **-ddoc-out-ostream <output-file> -** serializes the SignedDoc object and writes it to a DigiDoc file by using java.io.ObjectOutputStream. Note that the command is currently not being tested.
- **-ddoc-validate -** displays a DigiDoc file's content info and validates signature(s). Note that starting from the library's 3.8 version, warnings system is used, i.e. minor technical errors are printed out as warnings. See chapter "5.2.4.1 Validation status VALID WITH WARNINGS" for detailed information about warning situations.

  Parameter –libraryerrors can be added to the command to distinguish errors that are returned by the library.
- **-libraryerrors** – for testing purposes, may be used together with –ddoc-validate command. Enables to view the validation errors as they are returned by the library (otherwise, the utility program may transform specific errors to warnings; see also description under -ddoc-validate command). The errors are printed out with "LIBRARY-ERROR" prefix.
- **-ddoc-extract <data-file-id> <output-file> -** extracts DigiDoc file's content
- **-ddoc-rm-sig <signature-id> -** for testing purposes. Enables to remove a signature from DigiDoc container.
- **-ddoc-rm-df <data-file-id> -** for testing purposes. Enables to remove a data file from a DigiDoc container. Note that it is possible to remove data file only from an unsigned container.
- **-ddoc-list-keys -** experimental functionality. The command lists the tokens available on the device. Used for determining token's slot and label in case of signing or decrypting with HSM device (see also commands "-ddoc-sign-slot-label" and "-cdoc-decrypt-stream-slot-label")
- **-ddoc-sign-slot-label <slot-id> <label> <pin-code> [role/resolution] [country] [state] [city] [zip] [slot] [profile] -** experimental functionality. The command can be used when signing with a HSM device. The signature certificate on HSM device is determined by its slot ID number and the certificate object's label, both of the parameters are mandatory. The available tokens, their slots and labels can be found with command -ddoc-list-keys. **Note**: slot ID used in the current command refers to the actual ID value of the slot (not the sequence number of the certificate on device, as used in other JDigiDoc methods). Also, the signature certificate and private signature key have to be in the same slot and must have same label values (i.e. the label values of the certificate and private key objects are used to match the certificate with the appropriate private key).
- **–ddoc-calc-sign <cert-file> [role/resolution] [country] [state] [city] [zip]] [profile]** – for testing purposes or as a source of sample code. The command enables to calculate the hash to be signed, the actual signature calculation can then be made in an external system, e.g. with browser plug-in or by using DigiDocService web service. The signer's certificate (in PEM format) must be provided as input parameter.
- **–ddoc-add-sign-value <sign-val-file> <sign-id> -** for testing purposes or as a source of sample code. Enables to add a signature value (e.g. calculated externally via browser plug-in or DigiDocService web service) to a DigiDoc document.

6.2.1 Creating new DigiDoc files and signing

-ddoc-new [format] [version]

Creates a new digidoc container in the specified format and version. The current default format is DIGIDOC-XML, version 1.3. Alternative supported file format is BDOC, version 2.1.

Note: creating new DigiDoc files in older DigiDoc file formats SK-XML, DIGIDOC-XML 1.1, DIGIDOC-XML 1.2 and BDOC 1.0 is not supported.

-ddoc-add <input-file> <mime-type> [content-type]

Adds a new data file to a digidoc document. If digidoc container doesn't exist then creates one in the default format (DIGIDOC-XML 1.3). Data file can be added only to a container that is unsigned and doesn't contain an existing data file with the same name.

Input file (required) specifies the name of the original file, (it is recommended to include full path in this parameter; the path is removed when writing to DigiDoc container file, '/' and '\' characters are not allowed in the data file's name).

Mime type (required) represents the MIME type of the original file like "text/plain" or "application/msword".

Content type applies when using the DIGIDOC-XML format and reflects how the original files are embedded in the container: EMBEDDED_BASE64 (embedding binary data in base64 format) is supported and used by default.

-ddoc-add-mem <input-file> <mime-type> [content-type]

Analogous to -ddoc-add command but the data file's data is kept in a byte array in memory before adding it to the SignedDoc container.

-ddoc-sign <pin-code> [role/resolution] [country] [state] [city] [zip] [slot] [profile] [driver] [keystoreFile]

Adds a digital signature to the digidoc document. Note that adding signatures to DigiDoc files in older formats SK-XML, DIGIDOC-XML 1.1, DIGIDOC-XML 1.2 and BDOC 1.0 is no longer supported.

You can use the command with the following parameters:

pin code	Required. In case of Estonian ID cards, pin code2 is used for digital signing.
role/resolution	Role of the signer / resolution of the signer – as a single string, separated with a slash character. It is also possible to specify only the signer's role. At most one role/resolution value is allowed for a signature.
country	Country of origin. ISO 3166-type 2-character country codes are used (e.g. EE)
state	State or province where the signature is created
city	City where the signature is created
zip	Postal code of the place where the signature is created
slot	Identifier of the signer's signature certificate's and the accompanying private key's sequence number (counting from zero) among all signature certificates on an identity token. When operating for example with a single Estonian ID card (which contains one signature key) then the key can be found in slot 0 – which is used by default.

	The library makes some assumptions about pkcs11 drivers and card layouts: - you have on card signature and/or authentication keys - both key and certificate are in one slot - if you have many keys like 1 signature and 1 authentication key then they are in different slots - you can sign with signature key that has a corresponding certificate with "NonRepudiation" bit set. You may need to specify a different slot to be used when for example operating with multiple smart cards on the same system. In this case, the signature slots are counted as follows: - slot 0 – signature key of the 1st smartcard - slot 1 – signature key of the 2nd smartcard If the slot needs to be specified during signing, then the 5 previous optional parameters (manifest, country, state, city, zip) should also be filled first (either with the appropriate data or as " " for no value).
profile	Signature profile identifier. Used when adding a technical signature to a ddoc container. Technical signature is a signature with no OCSP confirmation and no timestamp value (analogous to bdoc "BES" profile). When creating a technical signature then the values of parameters "slot" and "profile" should be set to 0 and "BES" accordingly.
driver	Specifies the driver type that is used for signature creation (optional). Possible alternatives are: - PKCS11 – driver for singing with smart card, used by default. - PKCS12 – used when creating a technical signature with software token (PKCS#12 file). If signing with a software token (PKCS#12 file), then the appropriate changes must first be made in the configuration file (see section 3.5, subsection "Configuring software token usage").
keystoreFile	Specifies software token's PKCS#12 container's name. Used in case of signing with software token (i.e. the "driver" parameter of the current command has been set to "PKCS12").

-ddoc-out <output-file>

Stores the newly created or modified digidoc document in a file.

Sample commands for creating and signing DigiDoc files:

Sample: creating new DigiDoc file without signing, with default format and version (DIGIDOC-XML, version 1.3)

```
> java –jar jdigidocutil-*.jar -ddoc-new -ddoc-add c:\temp\test1.txt
text/plain -ddoc-out c:\temp\test1.ddoc

  Input:
  - c:\temp\test1.txt   - a data file to be added to container
  - text/plain          - mime type of the data file
```

```
  - c:\temp\test1.ddoc  - container to be created
```

Sample: creating new DigiDoc file with signing, with BDOC format and default profile (BDOC PROFILE TM)

```
> java –jar jdigidocutil-*.jar -ddoc-new BDOC -ddoc-add c:\temp\test1.txt
text/plain –ddoc-sign 12345 -ddoc-out c:\temp\test1.bdoc

Input:
  - BDOC                - specifies the digitally signed file's format
  - c:\temp\test1.txt   - a data file to be added to container
  - text/plain          - mime type of the data file
  - 12345               - id-card pin2
  - c:\temp\test1.bdoc  - container to be created
```

Sample: Signing an existing DigiDoc container (adding signatures)

```
> java –jar jdigidocutil-*.jar -ddoc-in c:\temp\test1.ddoc –ddoc-sign 67890
–ddoc-out c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc         - container to be signed
  - 67890                      - id-card pin2
  - c:\temp\test1.ddoc         - output (modified) digidoc container
```

Sample: Adding multiple data files to an existing unsigned DigiDoc container

```
> java –jar jdigidocutil-*.jar -ddoc-in c:\temp\test1.ddoc -ddoc-add
C:\temp\test3.txt text/plain -ddoc-add C:\temp\test4.txt text/plain –ddoc-
out c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc         - unsigned container to be read and modified
  - C:\temp\test3.txt          - first data file to be added
  - C:\temp\test4.txt          - second data file to be added
  - text/plain                 - mime type of the data files
  - c:\temp\test1.ddoc         - output(modified) digidoc container
```

Sample commands of creating technical signatures

Technical signature is a signature with no OCSP confirmation or a signature created with software token (PKCS#12 file). Note that verifying a signature that has no OCSP confirmation is expected to produce error message "Signature has no OCSP confirmation!". If the signature that is being verified was created with a software token (PKCS#12 file) then error message "Signer's cert does not have non-repudiation bit-set!" is produced.

Sample 1: signing an existing digidoc container with a technical signature, signature driver is defined in configuration file

```
> java –jar jdigidocutil-*.jar -ddoc-in c:\temp\test1.ddoc –ddoc-sign 67890
"" "" "" "" "" "" 0 "BES" –ddoc-out c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc  - unsigned container to be read and modified
  - 67890               - PIN
  - ""                  - empty strings for optional parameter values (role,
resolution, country, state, city, zip)
  - 0                   - signature slot
  - "BES"               - profile identifier of a technical signature
```

```
  - c:\temp\test1.ddoc  - output (modified) digidoc container

Sample 2: creating a new DigiDoc file, adding a data file and signing with
technical signature, using software token (PKCS#12 file)

> java –jar jdigidocutil-*.jar -ddoc-new -ddoc-add c:\temp\test.txt
text/plain –ddoc-sign 67890 “” “” “” “” “” “” 0 “BES” PKCS12
c:\test\pkcs12.pfx –ddoc-out c:\temp\request1.ddoc

Input:
- c:\temp\test.txt      - file to be added to container
- text/plain            - mime type of the file
- 67890                 - password of software token’s PKCS#12 container
- “”                    - empty strings for optional parameter values (role,
                       resolution, country, state, city, zip)
- 0                     - signature slot
- “BES”                 - profile identifier of a technical signature
- PKCS12                - identifier of PKCS12 module
- c:\test\pkcs12.pfx    - your software token’s PKCS#12 container file
- c:\temp\request1.ddoc - output digidoc container to be created
```

6.2.2 Reading DigiDoc files and verifying signatures

-ddoc-in <input-digidoc-file>

Specifies the input DigiDoc file name. It is recommended to pass the full path to the DigiDoc file in this parameter.

-ddoc-validate

Displays a DigiDoc file's content info and validates signature(s). Note that starting from the library's 3.8 version, warnings system is used, i.e. minor technical errors are printed out as warnings. See chapter "5.2.4.1 Validation status VALID WITH WARNINGS" for detailed information about warning situations.

Parameter –libraryerrors can be added to the command to distinguish errors that are returned by the library.

Returns:

- **List of all data files**, in format

DataFile <file identifier> (e.g. D0, D1…) file: <file name> (e.g. test1.txt) mime: <mime type> (e.g. text/plain) size: <file size in bytes> (e.g. 8)

- Returns signature **verification results** (if existing):

"Signature: S0 profile: <signature profile> - <signer's personal code, last name, first name> --> OK" or " --> ERROR"

- **Signature validation warning** (if present), in format:

WARNING: <error-code> - <warning message>

For example: WARNING: 178 - Test signature!

-ddoc-extract <data-file-id> <output-file>

Extracts the selected data file from the DigiDoc container and stores it in a file. **Data file id** represents the ID for data file to be extracted from inside the DigiDoc container.

Output file represents the name of the output file.

Sample commands for reading/validating/extracting from DigiDoc files:

```
Sample: validating existing DigiDoc file, signed
> java –jar jdigidocutil-*.jar -ddoc-in c:\Temp\test2.ddoc -ddoc-validate

Input:
- C:\temp\test2.ddoc    – the digidoc file to be validated
Returns:
  DataFile: D0 file: test2.txt mime: text/plain size: 8
  Signature: S0 profile: TM
  Signature: S0 profile: TM - MÄNNIK,MARI-LIIS,41109140240 --> OK


Sample: Extracting a data file from an existing DigiDoc file
> java –jar jdigidocutil-*.jar -ddoc-in c:\temp\test1.ddoc -ddoc-extract D1
c:\temp\test_ext.txt

Input:
- c:\temp\test1.ddoc    – the digidoc file to be extracted from
- D1                    - the data file ID to be extracted
- c:\temp\test_ext.txt - file for storing the extracted data
```

6.3 Encryption commands

Note: the utility program commands' parameters that are marked between "<" and ">" characters are mandatory and have to be specified. The parameters between "[" and "]" characters are optional.

- **-cdoc-in <input-encrypted-file>** - specifies the input encrypted document name
- **-cdoc-list -** displays the encrypted document data and recipients info
- **-cdoc-validate –** validates the encrypted document
- **-cdoc-test <input-file>** - tests whether input file is in valid DigiDoc format before encrypting it (checks for correct start and end tags). Note that the commands functionality is not tested periodically.
- -**cdoc-recipient <certificate-file> -** adds recipient to an encrypted document
- **-cdoc-encrypt-sk <input-file> <output-file> -** encrypts the input document; recommended for compatibility with other DigiDoc3 Crypto v3.8 and earlier, places the data file to be encrypted inside a new DigiDoc container. Alternatives are:
  - *-cdoc-encrypt <input-file> <output-file>* **-** used for encrypting small files, not recommended for compatibility with other DigiDoc software components
  - *-cdoc-encrypt-stream <input-file> <output-file>*- used for encrypting large files, not recommended for compatibility with other DigiDoc software components
- **-cdoc-decrypt-sk <pin> <output-file> -** decrypts the input file; recommended for compatibility with DigiDoc3 Crypto v3.8 and earlier, expects the encrypted input file to be in a DigiDoc container. Alternatives are:
  - **-***cdoc-decrypt <pin> <output-file>* **-** used for decrypting small files in any original format

- *-cdoc-decrypt-stream <input-file> <pin> <output-file>* - used for decrypting large files in any original format
- *–cdoc-decrypt-stream-recv <input-file> <pin> <output-file> <recipient>* - used for decrypting large files in any original format, uses the recipient name to locate the correct EncyptedKey element and the corresponding transport key to decrypt with. Note that the command is currently not being tested.
- *-cdoc-decrypt-pkcs12-sk <keystore-file> <keystore-passwd> <keystore-type> <output-file>* - decrypts document using pkcs12 sofware token, recommended for compatibility with other DigiDoc software components, expects the encrypted input file to be in a DigiDoc container.
- **-cdoc-decrypt-pkcs12 <keystore-file> <keystore-passwd> <keystore-type> <output-file>** - decrypts document using pkcs12 software token, used for decrypting small files in any original format
- *–cdoc-decrypt-pkcs12-stream-sk <input-file> <keystore-file> <keystore-passwd> <keystore-type> <output-file>* - used for decrypting large documents with pkcs12 software token. Recommended for compatibility with other DigiDoc software components, expects the encrypted file to be in DigiDoc format. Note that the command is currently not being tested.
- *–cdoc-decrypt-pkcs12-stream <input-file> <keystore-file> <keystore-passwd> <keystore-type> <output-file>* - decrypts document by using a pkcs12 software token, used for decrypting large files in any original format. Note that the command is currently not being tested.

- **-ddoc-list-keys -** experimental functionality. Lists the tokens available on the device. Used for determining token's slot and label in case of signing or decrypting with HSM device (see also commands "-ddoc-sign-slot-label" and "-cdoc-decrypt-stream-slot-label")
- **-cdoc-decrypt-stream-slot-label <input-file> <pin> <output-file> <slot> <label> -** experimental functionality. Can be used for decrypting data with HSM device. Decryption certificate on HSM device is determined by its slot ID number and the certificate object's label, both of the parameters are mandatory. Note that the slot ID used in the current method refers to the actual ID value of the slot (not the sequence number of the certificate on device, as used in other JDigiDoc methods). Also, the decryption certificate and accompanying private key have to be in the same slot and must have same label values (i.e. the label values of the certificate and private key objects are used to match the certificate with the appropriate private key).

6.3.1 Reading encrypted files

-cdoc-in <input-encrypted-file>

Specifies the input encrypted document name.

Input encrypted file (required) specifies the encrypted file.

-cdoc-list

Displays the encrypted data and recipient's info of an encrypted document just read in.

```
Sample: Displaying encrypted file's recipient info and data
> java –jar jdigidocutil-*.jar -cdoc-in c:\Temp\test1b.cdoc -cdoc-list

Input:
  - c:\temp\test1b.cdoc        – the encrypted file to be read
```

```
Returns:
  Encrypted document:

  EncryptedData type: http://www.isi.edu/innoes/iana/assignments/media-
  types/application/zip mime: http://www.isi.edu/in-
  noes/iana/assignments/media-types/application/zip

  algorithm: http://www.w3.org/2001/04/xmlenc#aes128-cbc
        FORMAT: ENCDOC-XML VER: 1.0
        LIBRARY: JDigiDoc VER: 2.7.0.30
        EncryptedKey Id: ID1 Recipient: MÄNNIK
              algorithm: http://www.w3.org/2001/04/xmlenc#rsa-1_5
                 CERT: SERIALNUMBER=41109140240, GIVENNAME=MARI-LIIS,
         SURNAME=MÄNNIK, CN=" MÄNNIK,MARI-LIIS,41109140240",
         OU=authentication, O=ESTEID, C=EE
        EncryptionProperties
                EncryptionProperty Name: LibraryVersion -->JDigiDoc|2.7.0.30
                EncryptionProperty Name: DocumentFormat --> ENCDOC-XML|1.0
                EncryptionProperty Name: Filename --> c:\temp\test1b.ddoc
                EncryptionProperty Name: OriginalSize --> 470
```

6.3.2 Encrypting files

-cdoc-test <input file>

Tests whether the input file is a valid digidoc document or not. It can be used to check the validity of the document before encrypting it.

Input file (required) specifies the original data file to be encrypted.

Returns:

- **Good ddoc**: <file name> - if the file is in valid DIGIDOC-XML format
- **Invalid ddoc**: <file name> - bad file begin – if the file does not start with '<?xml>' and '<SignedDoc>' tags
- **Invalid ddoc**: <file name> - bad file end – if the file does not end with '</SignedDoc>' tag

-cdoc-recipient <certificate-file> [recipient] [KeyName] [CarriedKeyName]

Adds a new recipient certificate and other metadata to an encrypted document. **Certificate file** (required) specifies the file from which the public key component is fetched for encrypting the data. The decryption can be performed only by using private key corresponding to that certificate.

Note: encryption should be done for the authentication certificates on all the recipient's valid identity tokens (e.g. the national ID-card and Digi-ID card used in Estonia), except of the Mobile-ID certificates. The input certificate files for encryption must come from the file system (DER and PEM encodings are supported). Possible sources where the certificate files can be obtained from include:

- Windows Certificate Store ("Other Persons")
- LDAP directories

- ID-card in smart-card reader

For example the certificate files for Estonian ID card owners' can be retrieved from a LDAP directory at ldap://ldap.sk.ee. The query can be made in following format through the web browser (IE): ldap://ldap.sk.ee:389/c=EE??sub?(serialNumber= xxxxxxxxxxx) where serial Number is the recipient's personal identification number, e,g.38307240240)

Other parameters include:

recipient	If left unspecified, then the program assigns a unique value to this attribute's value. This is later used as a command line option to identify the recipient whose key and smart card is used to decrypt the data. **Note:** Although this parameter is optional, it is recommended to pass on the entire CN value from the recipient's certificate as the recipient identifier here, especially when dealing with multiple recipients or using the –cdoc-decrypt-stream later on for decryption. For example if CN = MÄNNIK,MARI-LIIS,41110212444, then recipient = MÄNNIK,MARI-LIIS,41110212444 Otherwise, if left unspecified, then only the first part of the recipient's certificate's CN value is used (e.g. if CN = MÄNNIK,MARI-LIIS,41110212444, then recipient = MÄNNIK).
KeyName	subelement <KeyName> can be added to better identify the key object. Optional, but can be used to search for the right recipient's key or display its data in an application.
CarriedKeyName	subelement <CarriedKeyName> can be added to better identify the key object. Optional, but can be used to search for the right recipient's key or display its data in an application.

-cdoc-encrypt-sk <input-file> <output-file>

Encrypts the data from the given input file and writes the completed encrypted document in a file. **Recommended for providing cross-usability with other DigiDoc software components.**

This command places the data file to be encrypted in a new DigiDoc container. Therefore handling such encrypted documents later with other DigiDoc applications is fully supported (e.g. DigiDoc3 client).

Input file (required) specifies the original data file to be encrypted.

Output file (required) specifies the name of the output file which will be created in the current encrypted document format (ENCDOC-XML ver 1.0), with file extension **.cdoc**.

Note: There are also alternative encryption commands which are however **not recommended for providing cross-usability with other DigiDoc software components**:

-cdoc-encrypt <input-file> <output-file>

Encrypts the data from the given input file and writes the completed encrypted document in a file. Should be used only for encrypting **small** documents, **already in DIGIDOC-XML format.**

If using this command for encrypting documents not in DIGIDOC-XML format, then the receiver must also use the same JDigiDoc utility program for

opening/decrypting it, as cross-usability with other DigiDoc applications in this case is not supported.

Input file (required) specifies the original data file to be encrypted.

Output file (required) specifies the name of the output file which will be created in the current encrypted document format (ENCDOC-XML ver 1.0), with file extension **.cdoc**.

-cdoc-encrypt-stream <input-file> <output-file>

Encrypts the input file and writes to output file. Should be used only for encrypting **large** documents, **already in DIGIDOC-XML format.** Note that the command is not currently tested.

Input file (required) specifies the original data file to be encrypted.

Output file (required) specifies the name of the output file which will be created in the current encrypted document format (ENCDOC-XML ver 1.0), with file extension **.cdoc**.

If using this command for encrypting documents not in DIGIDOC-XML format, then the receiver must also use the same JDigiDoc utility program for opening/decrypting it, as cross-usability with other DigiDoc applications in this case is not supported.

Command line samples for encrypting documents:

Sample: encrypting small doc (DigiDoc compatible, original in any format)

```
> java –jar jdigidocutil-*.jar -cdoc-recipient c:\temp\Rcert.cer
MÄNNIK,MARI-LIIS,41110212444 -cdoc-encrypt-sk c:\temp\test_Small.txt
c:\Temp\test1.cdoc

Input:
- c:\temp\Rcert.cer     – the recipient's certificate file
- MÄNNIK,MARI-LIIS,41110212444 - the recipient's ID (= certificate's CN)
- c:\temp\test_Small.txt       – the input file to be encrypted
- c:\temp\test1.cdoc    - the encrypted file to be created
```

Sample: encrypting small doc (not DigiDoc compatible, unless original doc already in DIGIDOC-XML format)

```
> java –jar jdigidocutil-*.jar -cdoc-recipient c:\temp\Rcert.cer -cdoc-
encrypt c:\temp\test_Small.ddoc c:\Temp\test1.cdoc

Input:
- c:\temp\Rcert.cer     – the recipient's certificate file
- c:\temp\test_Small.ddoc      – the input file to be encrypted
- c:\temp\test1.cdoc    - the encrypted file to be created
```

Sample: encrypting large doc (not DigiDoc compatible, unless original doc already in DIGIDOC-XML format)

```
> java –jar jdigidocutil-*.jar -cdoc-recipient c:\temp\Rcert.cer -cdoc-
encrypt-stream c:\temp\test_Large.ddoc c:\Temp\test1.cdoc

Input:
- c:\temp\Rcert.cer     – the recipient's certificate file
- c:\temp\test_Large.ddoc      – the input file to be encrypted
- c:\temp\test1.cdoc    - the encrypted file to be created
```

Sample: testing original document format validity

```
> java –jar jdigidocutil-*.jar –cdoc-test c:\Temp\test1.ddoc

Input:
```

```
- c:\temp\test1.ddoc    – the digidoc container to be checked
Returns:
  Good ddoc: C:\temp\test1.ddoc
```

```
Sample: small doc, for multiple recipients
> java –jar jdigidocutil-*.jar -cdoc-recipient c:\temp\R1cert.cer -cdoc-
recipient c:\temp\R2cert.cer -cdoc-encrypt-sk c:\temp\test1.txt
c:\Temp\test2.cdoc

Input:
- C:\temp\R1cert.cer    – the 1st recipient's certificate file
- C:\temp\R2cert.cer    – the 2nd recipient's certificate file
- C:\temp\test1.txt     – the input file to be encrypted
- C:\temp\test1.cdoc    - the encrypted file to be created
```

6.3.3 Decrypting files

-cdoc-decrypt-sk <pin> <output-file> [slot(0)]

Decrypts and possibly decompresses the encrypted file just read in and writes to output file. Expects the encrypted file t**o be inside a DigiDoc container**.

Pin (required) represents the recipient's pin1 (in context of Estonian ID cards).

Output file (required) specifies the output file name.

Slot (optional) specifies sequence number (counting from zero) of the recipient's decryption certificate and accompanying private key on the identity token. Slot 0 is used by default. Note that the sequence number used in the current command may not be the same as the actual slot's ID.

Note: There are also alternative commands for decryption, depending on the encrypted file's format, size and the certificate type used for decrypting it.

***-cdoc-decrypt <pin> <output-file>* [slot(0)]**

Offers same functionality as -cdoc-decrypt-sk, should be used for decrypting **small** files (which do not need to be inside a DigiDoc container).

Pin (required) represents the recipient's pin1 (in contexts of Estonian ID cards).

Output file (required) specifies the output file name.

Slot (optional) specifies sequence number (counting from zero) of the recipient's decryption certificate and accompanying private key on the identity token. Slot 0 is used by default. Note that the sequence number used in the current command may not be the same as the actual slot's ID.

-cdoc-decrypt-stream <input-file> <pin> <output-file>

Offers same functionality as -cdoc-decrypt for decrypting documents, should be used for decrypting **large files** (which do not need to be inside a DigiDoc container). Note that the command is not currently tested.

Input file (required) specifies the original data file to be decrypted.

Pin (required) represents the recipient's pin1 (in contexts of Estonian ID cards).

Output file (required) specifies the output file name.

-cdoc-decrypt-pkcs12-sk <keystore-file> <keystore-passwd> <keystore-type> <output-file>

Offers same functionality as -cdoc-decrypt for decrypting documents, but using **software tokens** (PKCS#12 files). Expects the encrypted file t**o be inside a DigiDoc container**.

The following parameters are used with this decryption command:

<keystore-file>	Required. The path to the PKCS#12 file
<keystore-passwd>	Required. The password of the PKCS#12 file
<keystore-type>	Required. PKCS12
<output-file>	Required. The path and name of the encrypted output file

-cdoc-decrypt-pkcs12 <keystore-file> <keystore-passwd> <keystore-type> <output-file>

Offers same functionality as -cdoc-decrypt for decrypting documents, but using **software tokens** (PKCS#12 files). The encrypted file does not have to be inside a DigiDoc container.

The following parameters are used with this decryption command:

<keystore-file>	Required. The path to the PKCS#12 file
<keystore-passwd>	Required. The password of the PKCS#12 file
<keystore-type>	Required. PKCS12
<output-file>	Required. The path and name of the encrypted output file

Command line samples for decrypting documents:

```
Sample: decrypting small encrypted file, inside a DigiDoc container
> java –jar jdigidocutil-*.jar -cdoc-in c:\Temp\test1_small.cdoc -cdoc-
decrypt-sk 1234 c:\Temp\test1_d.ddoc

Input:
  - c:\Temp\test1_small.cdoc   – the encrypted file to be decrypted
  - 1234                       – the recipients pin1
  - C:\temp\test1_d.ddoc       - the decrypted file to be created

Sample: decrypting small encrypted file, in any original format
> java –jar jdigidocutil-*.jar -cdoc-in c:\Temp\test1_small.cdoc -cdoc-
decrypt 1234 c:\Temp\test1_d.ddoc

Input:
  - c:\Temp\test1_small.cdoc   – the encrypted file to be decrypted
  - 1234                       – the recipients pin1
  - C:\temp\test1_d.ddoc       - the decrypted file to be created


Sample: decrypting large encrypted file, in any original format
 > java –jar jdigidocutil-*.jar -cdoc-decrypt-stream
c:\Temp\test1_large.cdoc 1234 c:\Temp\test1_d.ddoc

Input:
```

```
  - c:\Temp\test1_large.cdoc    – the encrypted file to be decrypted
  - 1234                        – the recipients pin1
  - c:\temp\test1_d.ddoc        - the decrypted file to be created

Sample: decrypting, using PKCS#12 software token, in any original format
 > java –jar jdigidocutil-*.jar -cdoc-in c:\Temp\test1_small.cdoc -cdoc-
decrypt-pkcs12 c:\Temp\334836.p12d 12345pW PKCS12 c:\Temp\test1_d.ddoc

Input:
  - c:\Temp\test1_small.cdoc    – the encrypted file to be decrypted
  - c:\Temp\334836.p12d         - the PKCS#12 file
  - 12345pW                     - the PKCS#12 file's password
  - c:\temp\test1_d.ddoc        - the decrypted file to be created
```

7. National and cross-border support

7.1 National PKI solutions and support

7.1.1 Supported Estonian Identity tokens

Currently, JDigiDoc library is tested with the following Estonian ID tokens:

Token	Type	Description	Supported JDigiDoc functionality
EstEID 3.5 and 1.0	Certificate–based PKI smart cards	Different Estonian ID card versions	All JDigiDoc functionalities (authentication, signing, verification, encryption/decryption)
Digi-ID (since 2010)	Certificate–based PKI smart card	Estonian Digital ID card for use only in electronic environments	All JDigiDoc functionalities
Aladdin eToken Pro	Certificate–based PKI USB authenticator	Carrier for ID certificates issued to organizations.	**Note**: Supported and tested using the TempelPlus™ software, which is based on the JDigiDoc library.

7.1.2 Trusted Estonian Certificate Authorities

AS Sertifitseerimiskeskus (SK, http://sk.ee/en) functions as CA for all the Estonian ID tokens, maintains the electronic infrastructure necessary for issuing and using the ID cards, and develops the associated services and software.

SK issues the certificates and acts as Trusted Service Provider (TSP) for validation of authentication requests and digital signatures. SK maintains the following electronic services for checking certificate validity including:

- **OCSP validation service** (an RFC2560-compliant OCSP server, operating directly off the CA master certificate database and providing validity confirmations to certificates and signatures). There are two ways of getting access to the service:
  - having a contract with SK and accessing the service from a specific IP address(es) – as practiced **by companies/services**
  - by having certificate for accessing the service and sending signed requests - as used **by private persons** for giving digital signatures; registering for the service is required and service is limited to 10 signatures per month
- CRL-s (mainly for backward compatibility)
- LDAP directory service (containing all valid certificates)

7.1.2.1 Supported SK live certificate hierarchy chains

Note: no additional actions are needed for using the following CA and OCSP responder certificates with JDigiDoc - these certificate files have been:

- included in the JDigiDoc distribution
- registered in the JDigiDoc configuration file.

Certificate Common Name (CN)			Valid to	Description
JUUR-SK			26-Aug-2016	SK's 1st root certificate
	ESTEID-SK		13-Jan-2012	for ID cards issued until 2007

Certificate Common Name (CN)			Valid to	Description
		ESTEID-SK OCSP RESPONDER	*24-Mar-2005*	ESTEID-SK OCSP Responder
		ESTEID-SK OCSP RESPONDER 2005	*12-Jan- 2012*	ESTEID-SK OCSP Responder
	ESTEID-SK 2007		26-Aug-2016	for ID cards, Digi-ID and Mobile-IDs issued until 06.2011
		ESTEID-SK 2007 OCSP RESPONDER	*08-Jan-2010*	ESTEID-SK 2007 OCSP Responder
		ESTEID-SK 2007 OCSP RESPONDER 2010	*26-Aug-2016*	ESTEID-SK 2007 OCSP Responder
	EID-SK		08-May-2014	for all other personal certificates issued until 01.2007
		EID-SK 2007 OCSP RESPONDER	*15-May-2007*	EID-SK OCSP Responder
	EID-SK 2007		26-Aug-2016	for Estonian Mobile-IDs issued until 02.2011 and Lithuanian Mobile IDs issued until 06.2011
		EID-SK 2007 OCSP RESPONDER	*17-Apr- 2010*	EID-SK 2007 OCSP Responder
		EID-SK 2007 OCSP RESPONDER 2010	*26-Aug- 2010*	EID-SK 2007 OCSP Responder
	KLASS3-SK		05-May-2012	for organizational certificates issued until 10.2010
		KLASS3-SK OCSP RESPONDER	*05-Apr- 2006*	KLASS3-SK OCSP Responder
		KLASS3-SK OCSP 2006 RESPONDER	*27-Mar-2009*	KLASS3-SK OCSP Responder
		KLASS3-SK OCSP 2009 RESPONDER	*04-May- 2012*	KLASS3-SK OCSP Responder
	KLASS3-SK 2010		26-Aug-2016	for organizational certificates issued from 10.2010
		KLASS3-SK 2010 OCSP RESPONDER	*26-Aug- 2016*	KLASS3-SK 2010 OCSP Responder
EECCRCA			18-Dec- 2030	SK's 2nd root certificate
	ESTEID-SK 2011		18-Mar- 2024	for ID cards, Digi-ID and Mobile-IDs issued from 06.2011
	EID-SK 2011		18-Mar- 2024	for all other personal certificates issued from 06.2011

Certificate Common Name (CN)			Valid to	Description
	KLASS3-SK 2010		18-Mar-2024	for organizational certificates.
	SK OCSP 2011 RESPONDER		*18-Mar- 2024*	common OCSP responder for all certificates issued under EECCRCA

7.1.2.2 Supported SK test certificate hierarchy chains

Note: the following test certificates have been registered in the JDigiDoc configuration file but have not been included in the JDigiDoc distribution. In order to use the certificates with JDigiDoc, you need to copy the certificate files to a location referenced by the CLASSPATH (the files are accessible from https://installer.id.ee/media/esteidtestcerts.jar).

Note that the test certificates should not be used in live applications as the JDigiDoc library does not give notifications to the user in case of test signatures.

Certificate Common Name (CN)			Valid to	Description
Test JUUR-SK			27-Aug-2016	SK's 1st test root certificate
	TEST-SK		26-Aug-2016	for all test cards and certificates issued until 04.2011
		Test-SK OCSP RESPONDER 2005	*06-Apr-2012*	TEST-SK OCSP responder
	TEST of KLASS3-SK 2010		21-March-2025	for organizational test certificates
TEST EECCRCA			18-Dec-2030	SK's 2nd test root certificate
	TEST of ESTEID-SK 2011		07-Sep-2023	for test ID cards, Digi-ID and Mobile-ID certificates issued from 04.2011
	TEST of EID-SK 2011		07-Sep-2023	for all other test certificates issued from 04.2011
	Test SK OCSP RESPONDER 2011		*07-Sep-2024*	common OCSP responder for all test certificates issued under TEST-EECCRCA

For adding or removing CAs, OCSP responders or certificates, please refer to Section 3.5, **Configuring JDigiDoc,** under **Registering or removing CAs and OCSP responders.**

7.2 Cross-border support

The European Parliament and the Council adopted in December 1999 Directive 1999/93/EC on a Community framework for electronic signatures. The purpose of the Directive was to establish a legal framework for e-signatures and for certification service providers in the internal market. It has defined a qualified electronic signature as an advanced electronic

signature which is based on a qualified certificate and which is created by a secure-signature-creation device.

7.2.1 Trusted Service Provider Lists

Note: the support of using TSLs in JDigiDoc library is experimental and not tested.

The following is an overview of the features offered by JDigiDoc to support cross-border operability by using Trusted Service Provider Lists.

TSLs are used for creation and validation of digital signatures. TSL directory specified in the configuration file is used to retrieve the information about the Certification Service Providers.

During the creation of a digital signature, the TSL data is used according to the following principles (steps 3, 4, 5 apply when OCSP confirmations are required):

1. The issuing CA from the signer's certificate is retrieved.
2. The issuing CA is looked up from the TSL.
3. If the CA is found, then its corresponding OCSP Responder's info is retrieved
4. If the OCSP Responders is found, then an OCSP request is sent.
5. The OCSP Responder sends and signs the response, including its own certificate

During verification, the signer's CA and OCSP Responder info is checked against the TSL.

7.3 Interoperability testing

7.3.1 XAdES/CAdES Remote Plugtests

The XAdES/CAdES Remote Plugtests© Event specifies a number of test cases for checking the interoperability of the participants' implementations of Advanced Electronic Signatures for XML and CMS documents, also known as XAdES and CAdES.

The event evaluates (X-C)AdES interoperability by focusing on all the different XAdES forms standardized in ETSI TS 101 903 and ETSI 101 733, including (X-C)AdES-BES, (X-C)AdES-EPES, (X-C)AdES-T, (X-C)AdES-C, (X-C)AdES-X Type 1, (X-C)AdES-X Type 2, (X-C)AdES-XL and (X-C)AdES-A. More detailed information about the events can be found at the Remote Plugtest Portal: http:/www.etsi.org/plugtests/XAdes2/html/XAdES2.htm.

In the generation and cross-verification tests the participants are invited to generate a certain set of valid XAdES/CAdES signatures with certain characteristics (generation). The rest of participants are invited afterwards to verify these signatures (cross-verification).

In 2010, the main DigiDoc project coordinator AS Sertifitseerimiskeskus (SK) participated in the 6th Plugtests event (a partly anonymized report of the event is available at: http://xades-portal.etsi.org/pub/XAdES-CAdES%202010-Plugtests-External%20Final-Report-v1.0.pdf). The signature generated by SK through DigiDoc applications in XAdES XL form was tested for interoperability. The following properties of the signature needed to be verified by other participants in the test case for the XAdES-XL form:

- SigningTime
- SigningCertificate
- SignatureTimeStamp
- CompleteCertificateRefs
- CompleteRevocationRefs ett
- SigAndRefsTimeStamp

- CertificateValues
- RevocationValues

The test data generated by SK resulted in 2 successful and 4 failed verifications by the other participants.

7.3.2 ASiC Remote Plugtests

The compliance of the BDOC 2.1 containers to ASiC standard [6] was successfully tested in the course of ASiC Remote Plugtests© Event (19 November to 07 December 2012). The event aimed to ascertain the correctness and cross usability of ASiC-S and ASiC-E containers created by different participant organizations worldwide. The test cases included generation and cross-verification of ASiC containers with different features (including verification of incorrect files). Each participant chose the appropriate set of tests to be implemented.

The main BDOC 2.1 project coordinator, AS Sertifitseerimiskeskus (SK), participated in the event. Selected set of test cases was implemented with **Libdigidocpp** (C++) software library of DigiDoc system [20]. Tests that were carried out by SK included functionality that was also applicable for **BDOC 2.1** file format [2] - thus, the tests included generation and cross-verification of **ASiC-E** containers with **XAdES** signatures [4]. The following test cases were covered:

- testing ASiC-E container structure,
- testing ASiC-E container's syntactical conformance,
- testing correctness of XAdES-BES signature in ASiC-E container,
- negative tests of verifying ASiC-E container with invalid XAdES-BES signatures.

The implemented test cases did not cover generation and verification of signatures with time-marks or time-stamps (according to BDOC-TM and BDOC-TS profiles).

Results achieved by SK were as follows:

- test files that were generated by SK were successfully cross-verified by five different participants in 4 out of 6 implemented positive test cases (two of the test files were not verified by other participants).
- SK successfully cross-verified files generated by other three participants in 5 out of 6 implemented positive test cases (one of the files could not be verified because of incompatibility with BDOC 2.1 standard).
- SK successfully passed the negative test case which involved verification of incorrect test file.

Additional information about the ASiC Remote Plugtests event can be found from http://www.etsi.org/plugtests/ASiC/Home.htm.

Note that as BDOC 2.1 file format cross-usability tests are carried out for JDigiDoc and Libdigidocpp libraries (see also the next chapter) then the ASiC Remote Plugtests results are also reflected in JDigiDoc library.

7.3.3 DigiDoc framework cross-usability tests

Automated cross-usability tests of digitally signed and encrypted files are periodically carried out between different DigiDoc software libraries [20]:

- Cross-usability tests of digitally signed files in **DIGIDOC-XML 1.3** format (.ddoc files) are carried out between **JDigiDoc** and **CDigiDoc** software libraries.
- Cross-usability of **BDOC 2.1** (.bdoc or .asice) file format is tested between **JDigiDoc** and **Libdigidocpp** libraries.

- Cross-usability of encrypted file format **CDOC 1.0** is carried out between **JDigiDoc** and **CDigiDoc** software libraries.

The interoperability tests are executed through the **command line utility tools of the software libraries** (for example, in case of JDigiDoc library, the utility program which is described in chapter 6 of the current document).

7.3.4 Testing JDigiDoc API in JDigiDoc utility program

The JDigiDoc API's methods that are directly called out by JDigiDoc utility program are listed in the table below. Note that as the API is tested via the JDigiDoc utility program then the following functions are included in periodical tests and have been tested the most thoroughly.

JDigiDoc utility's command	Called JDigiDoc API method(s)
-ddoc-new	SignedDoc(String format, String version); SignedDoc.setProfile(String profile);
-ddoc-add <input-file> <mime-type>	SignedDoc(String format, String version); SignedDoc.addDataFile(File inputFile, String mime, String contentType);
-ddoc-add-mem <input-file> <mime-type>	SignedDoc(String format, String version); SignedDoc.readFile(File inFile); DataFile(String id, String contentType, String fileName, String mimeType, SignedDoc sdoc); DataFile.setBody(byte[] data); SignedDoc.addDataFile(DataFile df);
-ddoc-sign <pin-code>	SignedDoc.validate(boolean bStrong); SignedDoc.validateFormatAndVersion(); SignedDoc.countSignatures(); SignedDoc.verify(boolean checkDate, boolean demandConfirmation); SignatureProductionPlace(String city, String state, String country, String zip); ConfigManager.instance().getSignatureFactoryOfType(String sType); SignatureFactory.getType(); Pkcs12SignatureFactory.load(String storeName, String storeType, String passwd); SignatureFactory.getCertificate(int token, String pin); SignedDoc.prepareSignature(X509Certificate cert, String[] roles, SignatureProductionPlace adr); ConfigManager.instance().getStringProperty(String key, String def); Signature.setProfile(String profile); Signature.calculateSignedInfoDigest(); Signature.calculateSignedInfoXML(); SignatureFactory.sign(byte[] digest, int token, String pin, Signature sig); Signature.setSignatureValue(byte[] sigval); Signature.setHttpFrom(String s); Signature.getConfirmation();

JDigiDoc utility's command	Called JDigiDoc API method(s)
-ddoc-in <input-file>	DigiDocFactory. readSignedDocOfType(String fname, boolean isBdoc, ArrayList lerr);
-ddoc-in-stream <input-file>	SAXDigiDocFactory. readSignedDocOfType(String fname, boolean isBdoc, ArrayList lerr);
-ddoc-in-ostream <input-file>	-
-ddoc-out <output-file>	SignedDoc.writeToFile(File outputFile);
-ddoc-out-stream <output-file>	SignedDoc.writeToStream(OutputStream os);
-ddoc-out-ostream <output-file>	-
-ddoc-validate	SignedDoc.verify(boolean checkDate, boolean demandConfirmation); SignedDoc.hasFatalErrs(ArrayList lerrs); SignedDoc.countSignatures(); SignedDoc.countDataFiles(); DataFile.validate(boolean bStrong); Signature.verify(SignedDoc sdoc, ArrayList lerrs); Get methods of SignedDoc, Signature, KeyInfo classes.
-ddoc-extract <data-file-id> <output-file>	SignedDoc.countDataFiles(); SignedDoc.getDataFile(int idx); DataFile.getId(); DataFile.getBodyAsStream();
-check-cert	SignedDoc.readCertificate(File certFile); NotaryFactory.checkCertificate(X509Certificate cert);
-cdoc-in <input-file>	ConfigManager.instance().getEncryptedDataParser(); EncryptedDataParser.readEncryptedData(String fileName);
-cdoc-list	Get methods of EncryptedData, EncryptedKey and EncryptionProperty classes.
-cdoc-validate	EncryptedData.validate();
-cdoc-test <input-file>	-
-cdoc-encrypt <input-file> <output-file>	SignedDoc.readFile(File inFile); EncryptedData.setData(byte[] data); EncryptedData.setDataStatus(int status); EncryptedData.addProperty(String name, String content); EncryptedData.encrypt(int nCompressOption); EncryptedData.toXML();
-cdoc-encrypt-sk <input-file> <output-file>	ConfigManager.instance().getIntProperty(String key, int def); SignedDoc(String format, String version); SignedDoc.addDataFile(File inputFile, String mime, String contentType); SignedDoc.readFile(File inFile); DataFile.setBase64Body(byte[] data);

JDigiDoc utility's command	Called JDigiDoc API method(s)
	SignedDoc.toXML(); EncryptedData.setData(byte[] data); EncryptedData.setDataStatus(int status); EncryptedData.addProperty(String name, String content); EncryptedData.setMimeType(String str); EncryptedData.encrypt(int nCompressOption); EncryptedData.toXML();
-cdoc-encrypt-stream <input-file> <output-file>	ConfigManager.instance().getIntProperty(String key, int def); EncryptedData.addProperty(String name, String content); EncryptedData.encryptStream(InputStream in, OutputStream out, int nCompressOption);
-cdoc-recipient <certificate-file>	EncryptedData(String id, String type, String mimeType, String xmlns, String encryptionMethod); SignedDoc.readCertificate(File certFile); SignedDoc.getCommonName(String dn); EncryptedData.getNumKeys(); EncryptedKey(String id, String recipient, String encryptionMethod, String keyName, String carriedKeyName, X509Certificate recvCert); EncryptedData.addEncryptedKey(EncryptedKey key);
-cdoc-decrypt <pin> <output-file>	ConfigManager.instance().getSignatureFactoryOfType(String sType); SignatureFactory.getType(); Pkcs12SignatureFactory.load(String storeName, String storeType, String passwd); SunPkcs11SignatureFactory.init(String driver, String passwd, int nSlot); SignatureFactory.getAuthCertificate(int token, String pin); EncryptedData.getRecvIndex(X509Certificate cert); EncryptedData.decrypt(String driver, String keystoreFile, int nKey, int token, String pin);
-cdoc-decrypt-sk <pin> <output-file>	ConfigManager.instance().getSignatureFactoryOfType(String sType); SignatureFactory.getType(); Pkcs12SignatureFactory.load(String storeName, String storeType, String passwd); SunPkcs11SignatureFactory.init(String driver, String passwd, int nSlot); SignatureFactory.getAuthCertificate(int token, String pin); EncryptedData.getRecvIndex(X509Certificate cert); EncryptedData.decrypt(String driver, String keystoreFile, int nKey, int token, String pin); EncryptedData.getData(); ConfigManager.instance().getDigiDocFactory(); DigiDocFactory.readSignedDoc(String fileName);

JDigiDoc utility's command	Called JDigiDoc API method(s)
	SignedDoc.getDataFile(int idx); DataFile.getBodyAsStream();
-cdoc-decrypt-stream <input-file> <pin> <output-file>	ConfigManager.getEncryptedStreamParser(); EncryptedStreamParser.decryptStreamUsingTokenType(InputStream dencStream, OutputStream outs, int token, String pin, String tokenType, String pkcs12Keystore);
-cdoc-decrypt-stream-recv <input-file> <pin> <output-file> <recipient>	ConfigManager.instance().getEncryptedStreamParser(); EncryptedStreamParser.decryptStreamUsingRecipientName(InputStream dencStream, OutputStream outs, int token, String pin, String recipientName);
-cdoc-decrypt-pkcs12 <keystore-file> <keystore-passwd> <keystore-type> <output-file>	Pkcs12SignatureFactory.load(String storeName, String storeType, String passwd); Pkcs12SignatureFactory.getAuthCertificate(int token, String pin); EncryptedData.getRecvIndex(X509Certificate cert); EncryptedData.decryptPkcs12(int nKey, String keystore, String storepass, String storetype);
-cdoc-decrypt-pkcs12-sk <keystore-file> <keystore-passwd> <keystore-type> <output-file>	Pkcs12SignatureFactory.load(String storeName, String storeType, String passwd); Pkcs12SignatureFactory.getAuthCertificate(int token, String pin); EncryptedData.getRecvIndex(X509Certificate cert); EncryptedData.decryptPkcs12(int nKey, String keystore, String storepass, String storetype); EncryptedData.getData(); ConfigManager.instance().getDigiDocFactory(); DigiDocFactory.readSignedDoc(String fileName); SignedDoc.getDataFile(int idx); DataFile.getBodyAsStream();
-cdoc-decrypt-pkcs12-stream <input-file> <keystore-file> <keystore-passwd> <keystore-type> <output-file>	ConfigManager.instance().getEncryptedStreamParser(); EncryptedStreamParser.decryptStreamUsingTokenType(InputStream dencStream, OutputStream outs, int token, String pin, String tokenType, String pkcs12Keystore);
-cdoc-decrypt-pkcs12-stream-sk <input-file> <keystore-file> <keystore-passwd> <keystore-type> <output-file>	ConfigManager.instance().getEncryptedStreamParser(); EncryptedStreamParser.decryptStreamUsingTokenType(InputStream dencStream, OutputStream outs, int token, String pin, String tokenType, String pkcs12Keystore); EncryptedData.getData(); ConfigManager.instance().getDigiDocFactory(); DigiDocFactory.readSignedDoc(String fileName); SignedDoc.getDataFile(int idx); DataFile.getBodyAsStream();

8. JDigiDoc library's implementation notes

The following section describes properties of DIGIDOC-XML 1.3 and BDOC 2.1 files that are not strictly defined in the DIGIDOC-XML 1.3 [1] and BDOC 2.1 specification [2] but are used in JDigiDoc library's implementation (and also in other DigiDoc software libraries) of the file formats.

8.1 General implementation notes

Digital signature related notes:

1. One OCSP confirmation (time-mark) is allowed for each signature (due to security reasons and in order to maintain testing efficiency).
2. <Transforms> element is allowed in the signature since v3.10 of the library for interoperability purposes. The transformation algorithms are not applied during signature validation.
3. All data files in the container must be signed. All signatures in the container must sign all of the data files.
4. Exclusive Canonicalization method (http://www.w3.org/2001/10/xml-exc-c14n#) is supported during signature validation for better interoperability since v3.10 of the library.
5. During signature creation, it is checked that there is only one <ClaimedRole> element in the signature, which contains the signer's role and optionally the signer's resolution. If the <ClaimedRole> element contains both role and resolution then they must be separated with a slash mark, e.g. "role / resolution". Note that when setting the resolution value then role must also be specified.
6. During signature validation, at most two <ClaimedRole> elements are allowed for a signature.
7. Altering files in older formats SK-XML 1.0, DIGIDOC-XML 1.1 and 1.2 is not supported by the library. It is possible to validate and extract data files from these documents, but validation is expected to return error code about old DigiDoc file format. JDigiDoc utility program (identically to DigIDoc3 Client application) regards this validation error as a validation warning.
8. Opening and altering files in BDOC 1.0 format is not supported.

Certificate related notes:

1. Valid signatures (qualified electronic signatures) can be created with a certificate that has "Non-repudiation" value (also referred to as "Content Commitment") in its "Key usage" field. The requirement is based on the following sources:
    - ETSI TS 102 280 (V1.1.1): "X.509 V3 Certificate Profile for Certificates Issued to Natural Persons" [16]; chap. 5.4.3;
    - Profile of certificates issued to private persons by AS Sertifitseerimiskeskus: "Certificates on identity card of Republic of Estonia", version 3.3 [17]; appendix A.3.3;
    - Profile of certificates issued to legal entities by AS Sertifitseerimiskeskus: "Profile of institution certificates and Certificate Revocation Lists", version 1.3 [18]; chap. 3.2.2.
2. Signature can be created with a certificate that doesn't have "Non-repudiation" value in its "Key-Usage" field when specific parameters have been set but validation of

such signature will produce a respective error message and the signature is not considered as a qualified electronic signature.

3. During signature validation, it is checked that the validity periods of the signer's certificate and all the certificates in its CA chain include the signature creation time (value of the producedAt field in OCSP response).

8.2 DIGIDOC-XML 1.3 specific implementation notes

1. The only data file embedding mode that is supported, is CONTENT_EMBEDDED_BASE64 which means that the data file is included in the DigiDoc container in base64 encoding.
2. The nonce value's calculation in case of time-marking mechanism of DIGIDOC-XML 1.3 file format is implemented as follows:
   - the contents of <SignatureValue> element (i.e. the value without XML tags) is taken and decoded from base64 encoding;
   - digest of the value found in the previous step is calculated by using SHA-1 algorithm.
   - the digest value is included in the OCSP request's "nonce" field and must be present in the respective field of the OCSP response.
3. In case of DigiDocService web service [15] and DIGIDOC-XML 1.3 file format, JDigiDoc software library supports HASHCODE data file mode for intermediary ddoc files. The mode allows sending only the data file's digest value to the service, instead of embedding the whole data file to the container. In this case, it is possible to add larger data files than 4MB to the container (which would otherwise be the maximum data file size allowed in DigiDocService).
4. Embedding data files to the container as pure XML (EMBEDDED data file mode) and signing data files that are not included in the container (DETACHED data file mode) are not supported.
5. <DataFile> element's Id attribute value is set as "D<seq_no>" when adding the file to DigiDoc container. During verification, the Id attribute "DO" is also accepted as valid.
6. In case of DIGIDOC-XML 1.3 documents, the following validation errors are regarded as minor technical errors and are treated as validation warnings in jdigidoc utility program (identically to DigiDoc3 Client application):
   - <DataFile> element's xmlns attribute is missing.
   - <IssuerSerial><X509IssuerName> and/or <IssuerSerial><X509SerialNumber> element's xmlns attribute is missing.
7. It is possible to use JDigiDoc configuration file's parameter CHECK_OCSP_NONCE with DIGIDOC-XML 1.3 files, which, if set to "true", means that the presence of OCSP response's (the contents of <EncapsulatedOCSPValue> element) nonce value's ASN.1 prefix is not required during signature validation. Otherwise, it is required by RFC 2560 specification ("Online Certificate Status Protocol - OCSP") that the OCSP response's nonce value must have the corresponding ASN.1 prefix (OCTET STRING tag (04_{hex}) followed by the length of the nonce value in hexadecimal format). By default, the nonce value's ASN.1 prefix is not checked in order to support validation of DIGIDOC-XML 1.3 files created with JDigiDoc library's version below v3.7.

8.3 BDOC 2.1 specific implementation notes

Digital signature related notes:

1. The supported BDOC 2.1 signature profile is a XAdES-EPES profile with time-mark (BDOC-TM).

   The basic BDOC profile is XAdES-EPES as BDOC 2.1 specification requires that <SignaturePolicyIdentifier> element is present ([2], chap. 5.2).

   It is expected that a time-mark (OCSP confirmation) has been added to the signature as according to BDOC 2.1 specification ([2], chap. 6) a signature is not considered complete or valid without validation data from external services (i.e. a time-mark).

2. Signatures with time-stamps (i.e. BDOC TS profile) are not supported (including archive time-stamps) and will be implemented in the future. Validation data must be added to the signature with a time-mark (according to BDOC 2.1 specification [2], chap. 6.1).

3. In case of BDOC signatures with time-mark, the OCSP nonce field's value is calculated as follows:

   - the contents of <SignatureValue> element (i.e. the value without XML tags) is taken and decoded from base64 encoding;
   - digest of the value found in the previous step is calculated by using SHA-256 algorithm.;
   - the digest value found in the previous step and the digest algorithm that was used are transformed as defined by the following ASN.1 structure:

     ```
     TBSDocumentDigest ::= SEQUENCE {
        algorithm AlgorithmIdentifier,
        digest OCTET STRING
     }
     ```

   - the ASN.1 block value produced in the previous step is included in the OCSP request's "nonce" field and must be present in the respective field of the OCSP response.

4. In case of signing with ECC keys (by using ECDSA algorithm), concatenation method is used for creating signature value.

5. In case of BDOC 2.1 documents, SHA-256 hash function is used by default when calculating data file digests and the digest that is signed. In case of Estonian ID cards with certificates issued before 2011, the SHA-224 digest type will be automatically selected and used for calculating signature value's digest (the final digest that is signed), other options are not supported here. Note that other digest in the signature (e.g. data file digests, signer certificate's digest) are still calculated with SHA-256 (the default digest type).

6. When a hash function that is weaker than SHA-256 (or SHA-224 in the special case with pre-2011 ID-cards) has been used then a warning message about weak digest method is produced to the user. It is recommended to regard the error as a validation warning (identically to DigiDoc3. Client application and digidoc-tool.cpp utility program).

7. The signature policy document's hash value in <SigPolicyHash> element is checked during validation process (even though it is not mandatory according to BDOC 2.1 specification [2], chap. 5.2). The hash value must correspond to the hash value of the document that is located at https://www.sk.ee/repository/bdoc-spec21.pdf.

8. XML namespace prefixes are used in case of all XML elements (e.g. "asic:", "ds:", "xades:").
9. The <ds:Signature> element's Id attribute value is set to "S<seq_no>" during signature creation where sequence numbers are counted from zero. Other Id values that are used in the sub-elements of the <ds:Signature> element contain the signature's Id value as a prefix. During verification, different Id attribute values are also supported but are not tested periodically.

Container related notes:

1. BDOC 2.1 files are created with .bdoc file extension. Extensions .asice and .sce are supported and recognized only when reading in an existing BDOC 2.1 file which has the respective extension value.
2. Signatures are stored in META-INF/signatures*.xml files where '*' is a sequence number. Counting is started from one (note that Libdigidocpp library starts counting from zero). Two signtures*.xml files with duplicate names are not allowed in one container.
3. There can be only one signature in one signatures*.xml file due to BDOC format's legacy issues. Multiple signatures in one signatures*.xml file is not supported in order to maintain testing efficiency. The <ds:Signature> element's Id attribute values in different signatures*.xml files are generated in the form of "S<seq_no", the sequence numbers are always unique within one BDOC container.
4. The META-INF/manifest.xml file's OpenDocument version attribute value is "1.0" (instead of "1.2") as the results of ASiC plug-tests event shows that version 1.0 is used only. The requirement of the OpenDocument version attribute value comes from OpenDocument standard [5] which is referred to in ASiC standard [6].
5. Relative file paths are used, for example "META-INF/signatures*.xml" and "document.txt" instead of "/META-INF/signatures*.xml" and "/document.txt" to ensure better interoperability with third party applications when validating signatures. Advantage of using relative paths instead of absolute paths in the ASiC container was determined in the course of ASiC plug-tests (see also chap. "7.3.2 ASiC Remote Plugtests").
6. The ZIP container's comment field contains version number of the library that was used for creating the file. The value can be useful, for example, when trying to determine the origin of an erroneous file.
7. "mimetype" file is not compressed in the BDOC 2.1 file's ZIP container as the results of ASiC plug-tests event shows that this solution is most widely used.
8. It is not allowed to add two data files with the same name to a BDOC container as the signed data file must be uniquely identifiable in the container.
9. JDigiDoc does not support references to data files in BDOC container that start with slash character, i.e. value in the signatures*.xml file's <Reference> element's "URI" attribute and manifest.xml file's <file-entry> element's "full-path" attribute should not start with a slash.

Appendix 1: JDigiDoc configuration file

A sample jdigidoc.cfg file may consist of the following sections and possible entries:

- user-specific values to be always checked and possibly modified in purple
- optional and alternative settings in *blue*
- section headers in **green**
- # is indicating all out-commented parameters and additional notes

```
# JDigiDoc config file

# Signature processor settings
DIGIDOC_SIGN_IMPL        =       ee.sk.digidoc.factory.PKCS11SignatureFactory
# DIGIDOC_SIGN_IMPL = ee.sk.digidoc.factory.Pkcs12SignatureFactory
DIGIDOC_SIGN_IMPL_PKCS11 =       ee.sk.digidoc.factory.PKCS11SignatureFactory
DIGIDOC_SIGN_IMPL_PKCS12 =       ee.sk.digidoc.factory.PKCS12SignatureFactory
DIGIDOC_NOTARY_IMPL      =       ee.sk.digidoc.factory.BouncyCastleNotaryFactory
DIGIDOC_FACTORY_IMPL     =       ee.sk.digidoc.factory.SAXDigiDocFactory
DIGIDOC_TIMESTAMP_IMPL   =       ee.sk.digidoc.factory.BouncyCastleTimestampFactory
CANONICALIZATION_FACTORY_IMPL    =       ee.sk.digidoc.c14n.TinyXMLCanonicalizer
DIGIDOC_TSLFAC_IMPL      =       ee.sk.digidoc.tsl.DigiDocTrustServiceFactory
ENCRYPTED_DATA_PARSER_IMPL       =       ee.sk.xmlenc.factory.EncryptedDataSAXParser
ENCRYPTED_STREAM_PARSER_IMPL     =       ee.sk.xmlenc.factory.EncryptedStreamSAXParser

# Security settings
DIGIDOC_SECURITY_PROVIDER        =       org.bouncycastle.jce.provider.BouncyCastleProvider
DIGIDOC_SECURITY_PROVIDER_NAME   =       BC

# Big file handling
DIGIDOC_MAX_DATAFILE_CACHED      =       4096
# use this param if you want temp files in specific dir. You must have write access to it
# default is to use java.io.tempdir
# DIGIDOC_DF_CACHE_DIR=/tmp
DATAFILE_HASHCODE_MODE   =       FALSE

# Signature verification settings
CHECK_OCSP_NONCE         =       false

# default digest type for hash calculation - SHA-1, SHA-224, SHA-256, SHA-512
DIGIDOC_DIGEST_TYPE      =       SHA-256
# for .ddoc files, SHA-1 digest type is used always
BDOC_SHA1_CHECK  =       TRUE

# digidoc default profile for BDOC format
DIGIDOC_DEFAULT_PROFILE  =       TM
# TM = Qualified BDOC signature with time-marks
# alternative BDOC profile is BES

# PKCS#11 module settings - change this according to your signature device!!!
DIGIDOC_SIGN_PKCS11_DRIVER       =       opensc-pkcs11
# in linux and OSX environment: opensc-pkcs11.so
# for AID cards (GPK8000): DIGIDOC_SIGN_PKCS11_DRIVER = pk2privXAdES-XL.SCOK/SK/
DIGIDOC_SIGN_PKCS11_WRAPPER      =       PKCS11Wrapper

# log4j config file - change this!!!
DIGIDOC_LOG4J_CONFIG     =       ./log4j.properties

# OCSP responder URL - change this!!!
```

```
DIGIDOC_OCSP_RESPONDER_URL       =       http://ocsp.sk.ee
# Test OCSP responder URL (previously http://www.openxades.org/cgi-bin/ocsp.cgi)
#   DIGIDOC_OCSP_RESPONDER_URL = http://demo.sk.ee/ocsp

# connect timeout in milliseconds. 0 means wait forever
OCSP_TIMEOUT     =       30000

# By default, key-usage non-repudiation bit is checked for signature certificates
KEY_USAGE_CHECK  = TRUE

# Sign OCSP requests or not. Depends on your responder
SIGN_OCSP_REQUESTS       =       FALSE
# OCSP_SAVE_DIR = .
# The PKCS#12 file used to sign OCSP requests
# DIGIDOC_PKCS12_CONTAINER = <your-pkcs12-file-name>
# password for this key
# DIGIDOC_PKCS12_PASSWD = <your-pkcs12-passwd>
# serial number of your PKCS#12 signature cert.
# Use ee.sk.test.OCSPCertFinder to find this
# DIGIDOC_OCSP_SIGN_CERT_SERIAL = <your-pkcs12-cert-serial>

# CA certificates. Used to do a preliminary check of signer.
# use jar:// to get certs from classpath
# use forward slashes both on your Linux and other environments
DIGIDOC_CAS      =       1
DIGIDOC_CA_1_NAME        =       AS Sertifitseerimiskeskus
DIGIDOC_CA_1_TRADENAME   =       SK
DIGIDOC_CA_1_CERTS       =       17
DIGIDOC_CA_1_CERT1       =       jar://certs/EID-SK.crt
DIGIDOC_CA_1_CERT2       =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_CERT3       =       jar://certs/ESTEID-SK.crt
DIGIDOC_CA_1_CERT4       =       jar://certs/ESTEID-SK 2007.crt
DIGIDOC_CA_1_CERT5       =       jar://certs/JUUR-SK.crt
DIGIDOC_CA_1_CERT6       =       jar://certs/KLASS3-SK.crt
DIGIDOC_CA_1_CERT7       =       jar://certs/EECCRCA.crt
DIGIDOC_CA_1_CERT8       =       jar://certs/ESTEID-SK 2011.crt
DIGIDOC_CA_1_CERT9       =       jar://certs/EID-SK 2011.crt
DIGIDOC_CA_1_CERT10      =       jar://certs/KLASS3-SK 2010.crt
DIGIDOC_CA_1_CERT11      =       jar://certs/KLASS3-SK 2010 EECCRCA.crt

# SK-Test CA certs - only present if you have esteidtestcerts.jar in CLASSPATH. Should be
#commented out in case of live applications.
DIGIDOC_CA_1_CERT12      =       jar://certs/TEST-SK.crt
DIGIDOC_CA_1_CERT13      =       jar://certs/TEST EECCRCA.crt
DIGIDOC_CA_1_CERT14      =       jar://certs/TEST ESTEID-SK 2011.crt
DIGIDOC_CA_1_CERT15      =       jar://certs/TEST EID-SK 2011.crt
DIGIDOC_CA_1_CERT16      =       jar://certs/TEST KLASS3 2010.crt
DIGIDOC_CA_1_CERT17      =       jar://certs/test Juur-SK.crt

# OCSP responder certificates - change this!!!
# Note! if you add or remove some of these certificates, update the following number
# also pay attention to proper naming
DIGIDOC_CA_1_OCSPS       =       19

DIGIDOC_CA_1_OCSP1_CA_CN =       ESTEID-SK
DIGIDOC_CA_1_OCSP1_CA_CERT       =       jar://certs/ESTEID-SK 2007.crt
DIGIDOC_CA_1_OCSP1_CN    =       ESTEID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP1_CERT  =       jar://certs/ESTEID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP1_URL   =       http://ocsp.sk.ee
```

```
DIGIDOC_CA_1_OCSP2_CA_CN =      KLASS3-SK
DIGIDOC_CA_1_OCSP2_CA_CERT      =       jar://certs/KLASS3-SK.crt
DIGIDOC_CA_1_OCSP2_CN   =       KLASS3-SK OCSP RESPONDER
DIGIDOC_CA_1_OCSP2_CERT =       jar://certs/KLASS3-SK OCSP.crt
DIGIDOC_CA_1_OCSP2_CERT_1       =       jar://certs/KLASS3-SK OCSP 2006.crt
DIGIDOC_CA_1_OCSP2_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP3_CA_CN =      EID-SK
DIGIDOC_CA_1_OCSP3_CA_CERT      =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP3_CN   =       EID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP3_CERT =       jar://certs/EID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP3_URL  =       http://ocsp.sk.ee

# EID certificates (for example Mobile-ID certificates) issued since
# 20.01.2007 validity confirmation service
DIGIDOC_CA_1_OCSP4_CERT =       jar://certs/EID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP4_CN   =       EID-SK OCSP RESPONDER 2007
DIGIDOC_CA_1_OCSP4_CA_CERT      =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP4_CA_CN =      EID-SK 2007
DIGIDOC_CA_1_OCSP4_URL  =       http://ocsp.sk.ee

# Since 20.01.2007 issued ID-card certificates' validity confirmation service
DIGIDOC_CA_1_OCSP5_CN   =       ESTEID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP5_CERT =       jar://certs/ESTEID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP5_CA_CERT      =       jar://certs/ESTEID-SK 2007.crt
DIGIDOC_CA_1_OCSP5_CA_CN =      ESTEID-SK 2007
DIGIDOC_CA_1_OCSP5_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP6_CN   =       ESTEID-SK 2007 OCSP RESPONDER 2010
DIGIDOC_CA_1_OCSP6_CERT =       jar://certs/ESTEID-SK 2007 OCSP 2010.crt
DIGIDOC_CA_1_OCSP6_CA_CERT      =       jar://certs/ESTEID-SK 2007.crt
DIGIDOC_CA_1_OCSP6_CA_CN =      ESTEID-SK 2007
DIGIDOC_CA_1_OCSP6_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP7_CERT =       jar://certs/EID-SK 2007 OCSP 2010.crt
DIGIDOC_CA_1_OCSP7_CN   =       EID-SK 2007 OCSP RESPONDER 2010
DIGIDOC_CA_1_OCSP7_CA_CERT      =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP7_CA_CN =      EID-SK 2007
DIGIDOC_CA_1_OCSP7_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP8_CERT =       jar://certs/EID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP8_CN   =       EID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP8_CA_CERT      =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP8_CA_CN =      EID-SK 2007
DIGIDOC_CA_1_OCSP8_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP9_CERT =       jar://certs/ESTEID-SK OCSP 2005.crt
DIGIDOC_CA_1_OCSP9_CN   =       ESTEID-SK OCSP RESPONDER 2005
DIGIDOC_CA_1_OCSP9_CA_CERT      =       jar://certs/ESTEID-SK.crt
DIGIDOC_CA_1_OCSP9_CA_CN =      ESTEID-SK
DIGIDOC_CA_1_OCSP9_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP10_CERT =      jar://certs/SK OCSP 2011.crt
DIGIDOC_CA_1_OCSP10_CN  =       SK OCSP RESPONDER 2011
DIGIDOC_CA_1_OCSP10_CA_CERT     =       jar://certs/EECCRCA.crt
DIGIDOC_CA_1_OCSP10_CA_CN       =       EE Certification Centre Root CA
DIGIDOC_CA_1_OCSP10_URL =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP11_CA_CN       =       KLASS3-SK
DIGIDOC_CA_1_OCSP11_CA_CERT     =       jar://certs/KLASS3-SK.crt
```

```
DIGIDOC_CA_1_OCSP11_CN   =       SK Proxy OCSP Responder 2009
DIGIDOC_CA_1_OCSP11_CERT =       jar://certs/SK_proxy_OCSP_responder_2009.pem.cer
DIGIDOC_CA_1_OCSP11_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP12_CA_CN        =       KLASS3-SK
DIGIDOC_CA_1_OCSP12_CA_CERT      =       jar://certs/KLASS3-SK.crt
DIGIDOC_CA_1_OCSP12_CN   =       KLASS3-SK OCSP RESPONDER 2009
DIGIDOC_CA_1_OCSP12_CERT =       jar://certs/KLASS3-SK OCSP 2009.crt
DIGIDOC_CA_1_OCSP12_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP13_CERT =       jar://certs/ESTEID-SK OCSP.crt
DIGIDOC_CA_1_OCSP13_CN   =       ESTEID-SK OCSP RESPONDER
DIGIDOC_CA_1_OCSP13_CA_CERT      =       jar://certs/ESTEID-SK.crt
DIGIDOC_CA_1_OCSP13_CA_CN        =       ESTEID-SK
DIGIDOC_CA_1_OCSP13_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP14_CERT =       jar://certs/EID-SK OCSP.crt
DIGIDOC_CA_1_OCSP14_CERT_1       =       jar://certs/EID-SK OCSP 2006.crt
DIGIDOC_CA_1_OCSP14_CN   =       EID-SK OCSP RESPONDER
DIGIDOC_CA_1_OCSP14_CA_CERT      =       jar://certs/EID-SK.crt
DIGIDOC_CA_1_OCSP14_CA_CN        =       EID-SK
DIGIDOC_CA_1_OCSP14_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP15_CA_CN        =       KLASS3-SK 2010
DIGIDOC_CA_1_OCSP15_CA_CERT      =       jar://certs/KLASS3-SK 2010.crt
DIGIDOC_CA_1_OCSP15_CN   =       KLASS3-SK 2010 OCSP RESPONDER
DIGIDOC_CA_1_OCSP15_CERT =       jar://certs/KLASS3-SK 2010 OCSP.crt
DIGIDOC_CA_1_OCSP15_URL  =       http://ocsp.sk.ee
######## Test OCSP responders ###################
# Should be commented out in case of live applications.
DIGIDOC_CA_1_OCSP16_CERT =       jar://certs/TEST-SK OCSP 2005.crt
DIGIDOC_CA_1_OCSP16_CN   =       TEST-SK OCSP RESPONDER 2005
DIGIDOC_CA_1_OCSP16_CA_CERT      =       jar://certs/TEST-SK.crt
DIGIDOC_CA_1_OCSP16_CA_CN        =       TEST-SK
DIGIDOC_CA_1_OCSP16_URL  =       http://demo.sk.ee/ocsp

DIGIDOC_CA_1_OCSP17_CERT =       jar://certs/TEST SK OCSP 2011.crt
DIGIDOC_CA_1_OCSP17_CN   =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_CA_1_OCSP17_CA_CERT      =       jar://certs/TEST EECCRCA.crt
DIGIDOC_CA_1_OCSP17_CA_CN        =       TEST of EE Certification Centre Root CA
DIGIDOC_CA_1_OCSP17_URL  =       http://demo.sk.ee/ocsp

DIGIDOC_CA_1_OCSP18_CERT =       jar://certs/TEST SK OCSP 2011.crt
DIGIDOC_CA_1_OCSP18_CN   =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_CA_1_OCSP18_CA_CERT      =       jar://certs/KLASS3-SK 2010.crt
DIGIDOC_CA_1_OCSP18_CA_CN        =       KLASS3-SK 2010
DIGIDOC_CA_1_OCSP18_URL  =       http://demo.sk.ee/ocsp

DIGIDOC_CA_1_OCSP19_CA_CN        =       TEST of ESTEID-SK 2011
DIGIDOC_CA_1_OCSP19_CA_CERT      =       jar://certs/TEST ESTEID-SK 2011.crt
DIGIDOC_CA_1_OCS19_CN    =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_CA_1_OCSP19_CERT =       jar://certs/TEST SK OCSP 2011.crt
DIGIDOC_CA_1_OCSP19_URL  =       http://demo.sk.ee/ocsp

# Encryption settings
DIGDOC_ENCRYPT_KEY_ALG=AES
DIGIDOC_ENCRYPTION_ALOGORITHM=AES/CBC/PKCS7Padding
DIGIDOC_SECRANDOM_ALGORITHM=SHA1PRNG
DIGIDOC_KEY_ALOGORITHM=RSA/NONE/PKCS1Padding
```

Appendix 2: Signature types

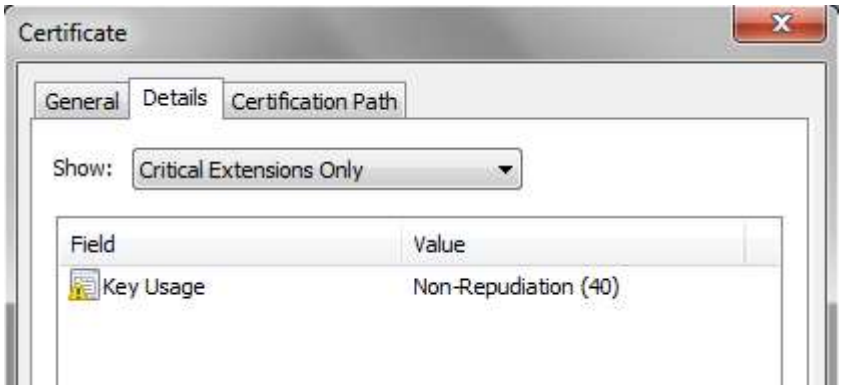
The signatures which are created can be either digital stamps, qualified electronic signatures or technical signatures depending on the certificate which is used for signing and whether OCSP confirmation is added or not.

Qualified electronic signature, i.e. ordinary digital signature

Qualified electronic signatures have the following characteristics:

- the certificate for signing has been issued to a private person,
- the signer's certificate has "Non-repudiation" value in its "Key usage" field (see also figure 1),
- the signature has OCSP confirmation.

Certificates which can be used for qualified electronic signature creation are stored on physical identity tokens: ID-card, Digi-ID, Mobile-ID or cryptostick.



A certificate with "Non-Repudiation" value in its "Key Usage" field

Digital stamp

Digital stamps are same as qualified electronic signatures, except of the certificate type that has been used for creating the signature. Digital stamps have the following characteristics:

- the certificate for signing is a "digital stamp" certificate issued to an organization (i.e. legal entity),
- the certificate has "Non-repudiation" value in its "Key usage" field (see also figure above),
- the signature has OCSP confirmation.

Digital stamp certificates are issued by AS Sertifitseerimiskeskus (SK) (see also https://www.sk.ee/en/services/Digital-stamp/), the certificates are stored on cryptosticks.

Technical signature

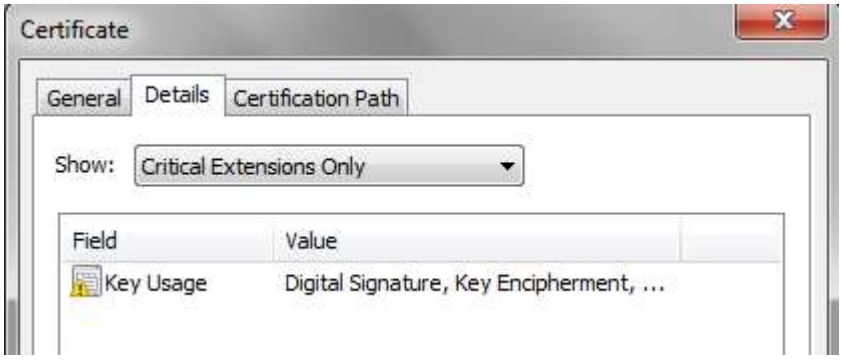
Technical signatures are signatures which have at least one of the following characteristics:

- the signer's certificate does not have "Non-repudiation" value in its "Key usage" field (see also figure below),
- OCSP confirmation has not been added to the signature.

Technical signatures can be created both by private persons and organizations.

Note: verification of a technical signature is expected to produce specific error message(s) depending on the signature's properties:

- technical signature with no OCSP confirmation is expected to produce error message "Signature has no OCSP confirmation!".
- technical signature which has been created with a certificate that doesn't have "Non-repudiation" value in its "Key usage" field is expected to produce error message "Signer's cert does not have non-repudiation bit set!".



A certificate with "Key Encipherment" value in its "Key Usage" field

Note that in the meaning of Estonian legislation (see [10]), qualified electronic signatures and digital stamps are equivalent to handwritten signatures whereas technical signatures are not.