JDigiDoc Programmer's Guide

Document Version: 3.6

Library Version: 3.6

Last update: 20.02.2012

1. Document versions

Document information	
Created on	20.02.2012
Reference	JDigiDoc Programmer's Guide
Receiver	Sertifitseerimiskeskus AS
Author	Veiko Sinivee, Kersti Üts, Kristi Uukkivi
Version	3.6

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

Table of contents

2. Introduction

This document describes JDigiDoc - the Java library for OpenXAdES/Digidoc system. It is a basic building tool for creating applications handling digital signatures, their verification and authentication. JDigiDoc covers all necessary functions like creation and verification of digitally signed files. The digitally signed files are created in „DigiDoc format" (with .ddoc or .bdoc file extensions), compliant to XML Advanced Electronic Signatures (XAdES), 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).

This document covers the following information about JDigiDoc:

- Section 2 introduces the OpenXAdES/DigiDoc framework, its general security model and formats available for digitally signed files.

- Section 3 gives an overview of the system requirements and configuration possibilities for JDigiDoc. It also describes the library's architecture and API for some of the most commonly used document signing and encryption tasks.

- Section 4 explains using the command line utility program for JDigiDoc, including sample use cases.

- Section 5 covers the currently supported tokens and CA's which have been tested with JDigiDoc and the current status of cross-border support for JDigiDoc.

- Appendix 1 provides a sample JDigiDoc.cfg configuration file.

2.1. About DigiDoc

JDigiDoc library forms a part of the wider OpenXAdES/DigiDoc system framework which offers a full-scale architecture for digital signature and documents, consisting of software libraries (C and Java), web service and end-user applications such as DigiDoc Portal and DigiDoc Client3 according to the following figure:
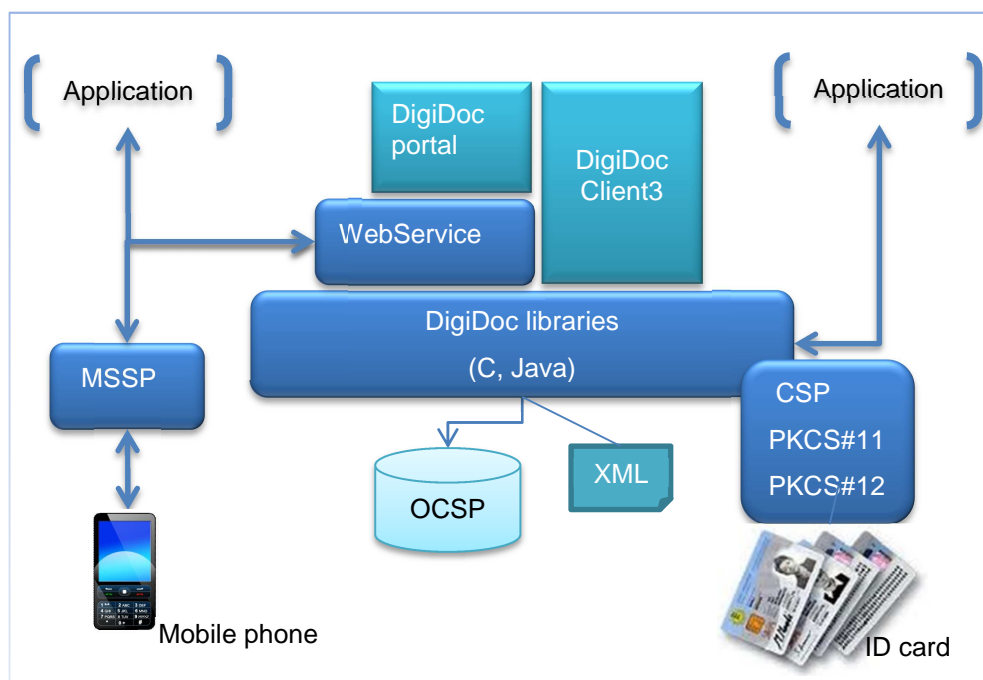
1 DigiDoc framework

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 format (current version DIGIDOC-XML 1.3) which is a profile of XAdES.

2.2. DigiDoc security model

The general security model of the DigiDoc and OpenXAdES ideology 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 is obtained in the format of Online Certificate Status Protocol (OCSP) response and stored within the signed document. Furthermore, (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".

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
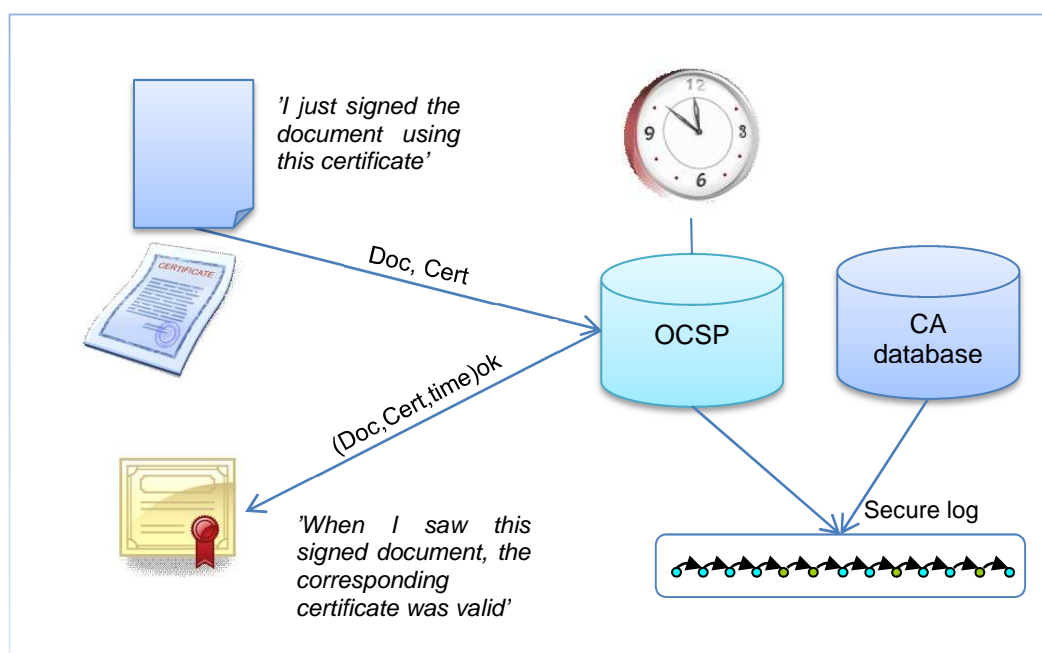
2 DigiDoc security model

2.3. Format of digitally signed file

The format of the digitally signed file is based on **ETSI TS 101 903** standard called **XML Advanced Electronic Signatures (XAdES)**. This standard provides syntax for digital signatures with various levels of additional validity information. JDigiDoc is implementing a subset of these standards.

In order to comply with the security model described above, the XAdES profile **XAdES-X-L** is used in the DigiDoc system but "**time-marks**" are used instead of "time-stamps" – signing (and certificate validation) time comes with OCSP response.

This profile:

- allows for incorporating following signed properties

  o Certificate used for signing

  o Signing time

  o Signature production place

  o Signer role or resolution

- incorporates full certificate validity information within the signature

o   OCSP response

o   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. Original files (which were signed) along with the signature(s), validation confirmation(s) and certificates are encapsulated within container with "SignedDoc" as a root element.
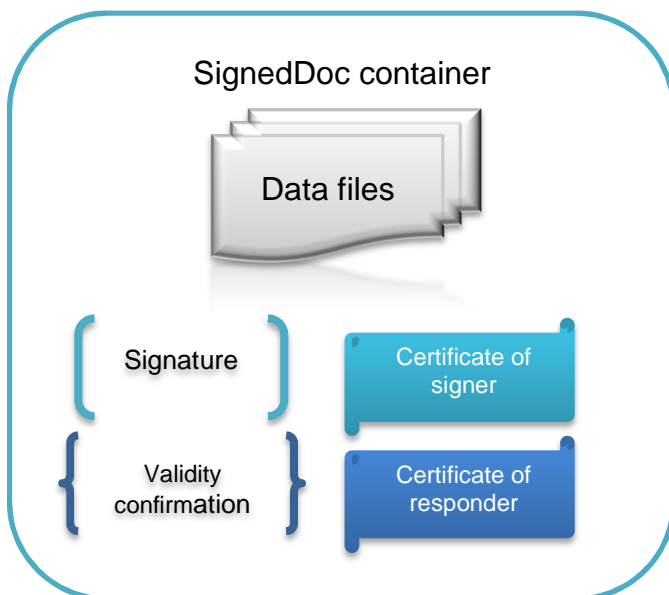
3 SignedDoc container

The library currently offers two main document formats to be used: **DIGIDOC-XML** and **BDOC.**

The DIGIDOC-XML document format (latest version 1.3) is fully conforming to XAdES standard (note however that not every single detail allowed in XAdES standard is supported). The BDOC format (latest version 1.0) is based on OpenDocument standard.

DigiDoc system uses file extension **.ddoc** or **.bdoc** to distinguish digitally signed files according to the described file format. Syntax of the .ddoc and .bdoc files is described in separate documents in detail, see [6], [9].

Main differences between DIGIDOC-XML (.ddoc) and the newer BDOC (.bdoc) formats are as follows:

Feature	DIGIDOC-XML (.ddoc)	BDOC (.bdoc)
Container Format	Single XML file	Zip-file
Data file adding mode	• EMBEDDED_BASE64 (embeds binary data in base64 encoding) • EMBEDDED (embeds pure text or XML) • DETACHED (adds only reference to an external file, no longer supported)	• BINARY

Contents of the container	- Data files embedded in the chosen mode - Signatures	- Data files in original format - Files for each signature - Files with metadata
Selecting profiles	N/A	Using signature profiles is a new feature with JDigiDoc and the BDOC format. BDOC profiles are based on the profiles defined by XAdES, offering various level of protection. **Note** that this feature for JDigiDoc is currently considered to be in beta phase status. a) Default profile is set to **BDOC_PROFILE_TM** b) A different default value can be set in the configuration file. c) A different value can be set also during each signature's creation
Currently supported profiles	N/A	BDOC_PROFILE_BES BDOC_PROFILE_T BDOC_PROFILE_CL **BDOC_PROFILE_TM** BDOC_PROFILE_TS
Selecting digest types for signature creation and other functions	N/A	Selecting digest types is a new feature with JDigiDoc and the BDOC format. It offers possibility to use hash algorithms belonging to the stronger SHA-2 family. Default value is **SHA-256.** A different default value can be set in the configuration file.
Currently supported digest types	Only **SHA-1** (set automatically)	SHA-1 SHA-224 **SHA-256** SHA-384 SHA-512

3. Overview

The following describes the JDigiDoc architecture, configuring possibilities and how to use it in Java programs.

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

- Creating files in supported DigiDoc formats (current defaults in **bold**):
    - o **DIGIDOC-XML 1.3**
    - o DIGIDOC-XML 1.2,
    - o DIGIDOC-XML 1.1
    - o **BDOC 1.0**

- Digitally **signing** the DigiDoc files using smart cards or other supported cryptographic tokens.

- Adding **time marks** (or timestamps) and **validity confirmations** to digital signatures using OCSP protocol.

- **Verifying** the digital signatures.

- Digital **encryption and decryption** of the DigiDoc files.

- For greater cross-border operability, the use of **Trust Service Status Lists** (TSL) and different XAdES-based **signature profiles** (with BDOC) is supported with JDigiDoc.

One of the design targets of this library is usability in various environments and possibility for the user to exchange parts of the library to other modules offering similar functionality by other means. Library encapsulates certain key functionality in separate modules and communicates with those modules over a fixed interface. The exchangeable modules have been implemented mostly as singleton – factory classes that implement the fixed Java interface for this functionality. There is always possibility to create a new module offering similar functionality and to change the configuration in order to make library use it.

As an example, the library does not assume the existence of a J2EE container; it can be used also in standalone Java programs.

As another example, the base library uses a separate module for creating new digital signatures with a smart card. This module uses a PKCS#11 driver to access the smart card. But third-party implementations exist (not part of JDigiDoc distribution) that implement the same interface using a cryptographic accelerator device (Hardware Security Module, HSM) for creating new digital signatures. Thanks to its modular design, the basic part of the library can still be used for creating and parsing digidoc files, regardless if the RSA signature is created by a smart card or other cryptographic devices.
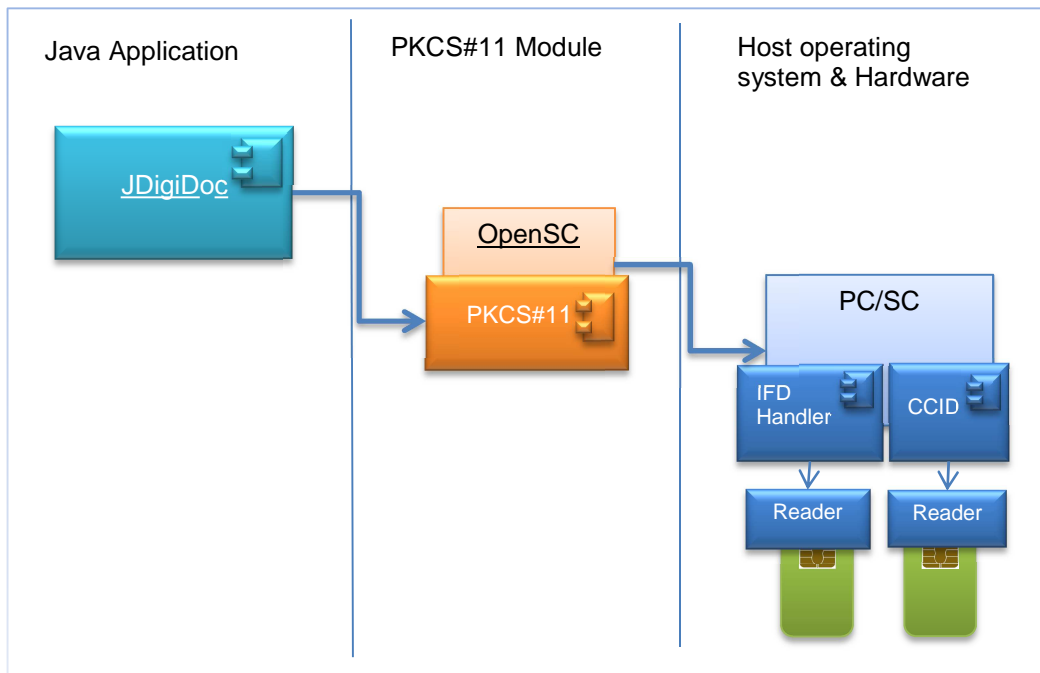
4 Sample JDigiDoc implementation using PKCS#11/ smart cards for digital signing

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

3.1. References and additional resources

[1] RFC2560	Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C., X.509 Internet Public Key Infrastructure: Online Certificate Status Protocol - OCSP. June 1999
[2] RFC3275	Eastlake 3rd D., Reagle J., Solo D., (Extensible Markup Language) XML Signature Syntax and Processing. (XML-DSIG) March 2002.
[3] ETSI TS 101 903	XML Advanced Electronic Signatures (XAdES). February 2002
[4] XML Schema 2	XML Schema Part 2: Data types. W3C Recommendation 02 May 2001 (http://www.w3.org/TR/xmlschema-2/)
[5] DSA	Estonian Digital Signature Act
[6] DigiDoc format	DigiDoc file format (http://www.id.ee/files/digidoc_vorming_1_3.pdf)

]7] XML-ENC	http://www.w3.org/TR/xmlenc-core/
[8] ISO/IEC 26300 :2006 Information technology	Open Document Format for Office Applications (OpenDocument) v1.0
[9] BDOC 1.0	http://www.evs.ee/tooted/evs-821-2009
[10] TSL - ETSI TS 102 231 ver. 3.1.2 (2009-12)	Electronic Signatures and Infrastructures (ESI); Provision of harmonized Trust-service status information
[11] DigiDocService Specification	EN: http://sk.ee/upload/files/DigiDocService_spec_eng.pdf ET: http://www.sk.ee/upload/files/DigiDocService_spec_est.pdf
[12] Release notes	JDigiDoc library's release notes

3.2. Terms and acronyms

BDOC (.bdoc)	Term is used to denote a profile of XAdES and container packaging rules bases on OpenDocument standard. The BDOC Basic Profile (based on XAdES-BES) 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. BDOC profiles providing means for certificate validation and trusted time-stamp data with the signature are BDOC with time-marks and BDOC with time-stamps. Both profiles are compliant to XAdES-X-L. • BDOC with time-marks uses OCSP protocol for time-marking and certificate validity confirmation. • BDOC with time-stamps is used in case OCSP is not replacing need for additional trusted time-stamps from external Time-Stamping Authority. The BDOC container format bases on OpenDocument standard, which foresees a ZIP container with file structure inside. The file extension for BDOC file format is ".bdoc", MIME-type is "application/bdoc".
CDOC (.cdoc)	The term denotes a format of an encrypted DigiDoc document that is based on XML-ENC profile.
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 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".
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.
TSL	The EU Trusted Service List, Supervision/Accreditation Status List of certification/services from Certification Service Providers which are supervised/accredited by the referenced EU Member State for compliance with the relevant provisions laid down in the eSignatures Directive 1999/93/EC. The set of 27 TSLs, each one issued by one EU Member State shall constitute a core piece in the European framework for mutual recognition of electronic signatures. Its structure is defined in XML and ASN.1. and contents include: - Preface with details of the assessment scheme and the TSL itself - Details on each entity providing the service(s). - Current details on each service provided by a certain entity. - Historical details on each service provided by a certain entity reporting status changes. In accordance with the ETSI TS 102 231 standard, the compiled list (the European Commission list of the locations where the Trusted Lists are published as notified by Member States) is available on a secure web-site in two formats: • Compiled list in a human readable format PDF (https://ec.europa.eu/information_society/policy/esignature/trusted-list/tl-hr.pdf) • Compiled list in a format suitable for automated (machine) processing XML (https://ec.europa.eu/information_society/policy/esignature/trusted-list/tl-mp.xml)

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. Dependencies

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

Base Component	Description
Java Platform	JDK/JRE 1.3.1 or newer
Bouncy-Castle cryptographic library	Used in cryptographic operations. This library was chosen as it is a freeware module. Note that you can use BouncyCastleNotaryFactory class to replace the IAIKNotaryFactory module. It offers the equivalent OCSP functionality but requires only BouncyCastle library (freeware) and thus you need no proprietary library's for using JDigiDoc. However, this only works with BouncyCastle 1.20 or newer and therefore the IAIK JCE library used to be a requirement earlier.
IAIK JCE cryptographic library	Required if choosing the IAIKNotaryFactory class as the module handling OCSP confirmations. Can also be replaced with BouncyCastle library 1.20 or newer.
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
DOMXmlCanonicalizer	ee.sk.digidoc.factory.DOMCanonicalizationFactory Uses Apache XML Security for XML canonicalization. Unfortunately works only with an older version of Apache Xmlsec and depends also on a number of older versions of

Apache Xerces	Xerces and Xalan.
	Xerces XML parser – required for the Apache XML Security library. Unfortunately cannot be replaced with JAXB or another parser because it is hard-coded in Apache XML Security library. Xerces could also be used for normal xml parsing and thereby needing only one xml parser, but we don't want to make this a requirement. The system default XML parser can be still used for normal digidoc parsing as long as it supports SAX interface and in addition to that, Xerces will be needed for canonicalization
Xalan	Xalan - version 2.2D13 or newer. Required for the Apache XML Security library.
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

3.4. 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
bcmail-jdk*-146.jar **bcprov-jdk*-146.jar** **bcpg-jdk*-146.jar** **bctsp-jdk*-146.jar** **or newer**	Bouncy-Castle cryptographic library, a Java implementation of cryptographic algorithms. Choose the releases according to your JDK version, e.g. for JKD 5.0 → bcmail-jdk15-146.jar, etc. Latest releases available from: http://www.bouncycastle.org/latest_releases.html
iaik_jce.jar	IAIK Java cryptographic library. Only needed if you are using also IAIKNotaryFactory module. Latest releases available from (licensed): https://jce.iaik.tugraz.at/sic/Download
xmlsec.jar + ***xalan.jar,*** + ***xercesImpl.jar,*** + ***xml-apis.jar,*** ***xmlParserAPIs.jar***	Apache XML Security, Xalan and Xerces libraries. Needed for using DOM canonicalizer. Latest releases available from: http://santuario.apache.org/download.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	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.
• *for Windows:* + *opensc-pkcs11.dll* + *Pkcs11Wrapper.dll*	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*	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

3.5. Configuring JDigiDoc

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

This file can be inside the JDigiDoc.jar library or in any other location referenced by CLASSPATH. It can also be stored in another location, not referenced by CLASSPATH, but then the full filename should be passed to the ConfigManager.init() method.

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**: In this version provides only verification of timestamps)
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)
CRL_FACTORY_IMPL	Module for handling CRLs. ***ee.sk.digidoc.factory.CRLCheckerFactory*** (implementation module for downloading CRLs)
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 ***4096*** *(4 KB)*
DIGIDOC_DF_CACHE_DIR	Specifies directory for storing temporary files. Default value is read from system parameter java.io.tmpdir ***/tmp***

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.

Therefore, as a feature of the **BDOC** 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 new signatures (this hash will be sent with the OCSP request) ***SHA-256*** # SHA-1, SHA-224, SHA-384, SHA-512
DIGIDOC_DEFAULT_DIGEST	Specifies the default digest type for all other digests (e.g. the hash of the data files, which serves as input for creating a signature) ***SHA-256*** # SHA-1, SHA-224, SHA-384, SHA-512

Additional notes on default digest type usage:

- for the DIGIDOC-XML format, SHA-1 will still the default digest type and cannot be altered.

- using SHA-512 hash function with JDigiDoc has so far not been tested as fully as the other hash functions of the SHA-2 family

- for Estonian ID cards with certificates issued before 2011, the SHA-224 digest type will be automatically selected and used when signing a document in BDOC format; other options are not being supported there.

Default profile for BDOC

Profiles are a feature for of BDOC document format and are based on the profiles defined by XAdES, offering various level of protection. For details on the specific options provided, see Section 5.2.2 on supported BDOC format profiles.

Parameter	Description
DIGIDOC_DEFAULT_PROFILE	Specifies the default profile to be used when creating a new document in BDOC format **TM** Profile based on XAdES-X-L, using time marks # BES, T, CL, TS

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 environment)* *# opensc-pkcs11.dll (used in Windows environments)*
DIGIDOC_SIGN_PKCS11_WRAPPER	A wrapper library which makes PKCS#11 modules available from within Java. ***PKCS11Wrapper***
DIGIDOC_VERIFY_ALGORITHM	The algorithm used when verifying the digital signatures, in format "algorithm/mode/padding". Since digital signing using Estonian ID cards is implemented using RSA keys, the default value is set to RSA: ***RSA//***

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 SignatureLogging.properties file **./jdigidoc/src/main/resources/SignatureLogging.properties**

TSL directory

TSL directory is only for cross-border support and is currently not tested. See section "5.2 Cross-border support" for more information.

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

Parameter	Description
DIGIDOC_TSL_DIR	All the files from this directory are read in and parsed to create a TSL List **./jdigidoc/src/main/tsl**

OCSP responder settings

This DIGIDOC_OCSP_RESPONDER_URL setting applies to your default OCSP responder address in following cases:

- When no other OCSP responder address for the CA is found in the TSL list

- Or 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://www.openxades.org/cgi-bin/ocsp.cgi) is for the OpenXAdES OCSP Testing Responder which can be used for testing purposes. For more information on using the OpenXAdES testing environment, please refer to http://www.openxades.org/tryitout.html.

For example, the real-life OCSP Responder address to be used for Estonian ID cards would be http://ocsp.sk.ee.

Parameter	Description

DIGIDOC_OCSP_RESPONDER_URL	OSPC Responder used if no other address is found for the CA in the locally registered or TSL entries. ***http://www.openxades.org/cgi-bin/ocsp.cgi***
OCSP_TIMEOUT	Connect timeout in milliseconds; 0 means wait forever **30000**

Mobile-ID signing

These settings will apply when using a Mobile-ID over the DigiDocService web service (DDS) and Mobile Signature Service Provider (MSSP) for digital signing.

The default address provided for the DigiDocService URL (https://www.openxades.org:8443) is a test address for using in the OpenXAdES testing environment. For more information on using the OpenXAdES testing environment for M-ID, please refer to http://www.openxades.org/ddsregisteruser.

Parameter	Description
DDS_URL	The DigiDocService URL, the default value provided is usable for testing: ***https://www.openxades.org:8443*** ***(test)*** *# https://digidocservice.sk.ee* *(live)*
DDS_POLLFREQ	Frequency in seconds at which a SOAP messages will be sent to the DigiDocService to query M-ID signing process status ***5***
DDS_TRUSTSTORE	Specifies the Java trust store file where the trusted HTTPS based DigiDoc web service server certificates are added (comes bundled with the JDigiDoc) ***./jdigidoc/src/main/resources/openxades.jks***
DDS_STOREAPASS	Password for the provided Java trust store file ***openxades***
DDS_STORETYPE	Type of the provided Java trust store file ***JKS***

HTTP proxy settings*

 *only necessary if using a proxy to access internet.

In the configuration provided with JDigiDoc, these parameters have been commented out.

If using a proxy server to for routing HTTP requests, remove the #-tags and enter values for following parameters:

Parameter	Description
DIGIDOC_PROXY_HOST	Specifies the proxy hostname, e.g. ***proxy.example.net***
DIGIDOC_PROXY_PORT	Specifies the proxy port, e.g. ***8080***

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).

By default, this parameter value is set to FALSE – i.e. the OCSP requests will not be signed.

If setting this to TRUE, 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*
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 the test program ee.sk.test.OCSPCertFinder for finding it.

TSL and CA certificates settings

You can select whether you want to use only the TSL-s or also use your 'local' CA certificates directly registered in the configuration file when creating and verifying digital signatures (e.g. since test CA certificates will generally not be included .in the official TSLs, you will need to register them manually in your configuration file).

For using also the CA certificates registered in the configuration file, the following parameter must be set to TRUE:

Parameter	Description
DIGIDOC_USE_LOCAL_TSL	False - for using only TSLs from DIGIDOC_TSL_DIR True - for using both TSLs and the registered CA certs **TRUE**

CA certificates

The CA certificates will be used to do a preliminary check of the signer's certificates.

You should provide here the following details about each 'local' CA that have not been included in your TSL list.

Note: 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)

Parameter	Description
DIGIDOC_CAS	Number of 'local' CAs registered in your configuration file *3*
DIGIDOC_CA_1_NAME … DIGIDOC_CA_*n*_NAME	Name of the registered CA's. The currently registered CA's in JDigiDoc are: *DIGIDOC_CA_1_NAME = AS Sertifitseerimiskeskus* *DIGIDOC_CA_2_NAME = Nacionalinis sertifikavimo centras* *DIGIDOC_CA_3_NAME = VI Registru Centras RCSC* (The number of entries is according to DIGIDOC_CAS)
DIGIDOC_CA_1_TRADENAME … DIGIDOC_CA_*n*_TRADENAME	Trade name for the CA *DIGIDOC_CA_1_TRADENAME = SK* *DIGIDOC_CA_2_TRADENAME = NSC* *DIGIDOC_CA_3_TRADENAME = RCSC* (The number of entries is according 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: *For SK: DIGIDOC_CA_1_CERTS = 13* *For NSC: DIGIDOC_CA_2_CERTS = 7* *For RCSC: DIGIDOC_CA_3_CERTS = 3*

	(The number of entries is according to DIGIDOC_CAS)
DIGIDOC_CA_1_CERT1 … DIGIDOC_CA_*n*_CERT*n*,	Location of all certificates for each CA (The number of entries is according 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. *jar://certs/TEST EECCRCA.crt*

OCSP responder certificates

When using your 'local' CAs 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 = 14*
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. *TEST of EE Certification Centre Root CA*
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/TEST EECCRCA.crt*
DIGIDOC_CA_1_OCSP1_CN … DIGIDOC_CA_*n*_OCSP*n*_CN	Name of the specific OCSP responder, e.g. *TEST of SK OCSP RESPONDER 2011*
DIGIDOC_CA_1_OCSP1_CERT … DIGIDOC_CA_*n*_OCSP*n*_CERT	Location of the OCSP responder's certificate, e.g. *jar://certs/TEST SK OCSP 2011.crt*
DIGIDOC_CA_1_OCSP1_URL … DIGIDOC_CA_*n*_OCSP*n*_URL	Address for the OCSP responder, e.g. *http://www.openxades.org/cgi-bin/ocsp.cgi*

OCSP or CRL selectors

You can choose whether to use Certificate Revocation Lists (CRLs) or Online Certificate Status Protocol (OCSP) confirmations for obtaining the singer's certificate revocation status. **Note** that using OCSP has the primary benefit of requiring less network bandwidth and enabling real-time status checks over using CRLs.

Parameter	Description
DIGIDOC_CERT_VERIFIER	*OCSP* OCSP confirmations are selected by default
DIGIDOC_SIGNATURE_VERIFIER	*OCSP* OCSP confirmations are selected by default
DIGIDOC_TSA_URL	Address for your timestamp service provider if using time stamping *http://tsp.iaik.tugraz.at/tsp/TspRequest*

CRL settings*

*only necessary if using CRLs to verify the signer's certificate validity. Will not be used if using OCSP confirmations instead.

Note: using CRLs is being discouraged since providing less secure and efficient means compared to OCSP.

If using CRLs, you should set your DIGIDOC_CERT_VERIFIER and DIGIDOC_SIGNATURE_VERIFIER to CRL and enter details about accessing your CRL data using the following parameters:

Parameter	Description
CRL_USE_LDAP	Whether using LDAP or not for accessing CRLs **FALSE**
CRL_FILE	File name of the CRL file **esteid.crl**
CRL_URL	Location of the CRL file **http://www.sk.ee/crls/esteid/esteid.crl**
CRL_SEARCH_BASE	**cn=ESTEID-SK,ou=ESTEID,o=AS Sertifitseerimiskeskus,c=EE**
CRL_FILTER	**certificaterevocationlist;binary=*)**
CLR_LDAP_DRIVER	LDAP settings **com.ibm.jndi.LDAPCtxFactory**
CRL_LDAP_URL	LDAP settings **ldap://194.126.99.76:389**
CRL_LDAP_ATTR	LDAP settings **certificaterevocationlist;binary**
CRL_PROXY_HOST	CRL proxy host **cache.eypsise**
CRL_PROXY_PORT	CRL proxy port **8080**

Encryption settings

Parameter	Description
DENC_COMPRESS_MODE	Compression mode of the original data before encryption. Possible values are 0 – always compress, 1 – never compress, 2 – best effort (compression is used only if it results in reduced data size). # 0, # 1, # 2 Note that in jdigidoc utility program, "always compress" mode is used by default.

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 SignatureLogging.properties –

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

# JDigiDoc loggers
log4j.logger.ee.sk.utils.ConfigManager=DEBUG, output
log4j.logger.ee.sk.digidoc.DigiDocException=DEBUG, output
log4j.logger.ee.sk.digidoc.factory.IAIKNotaryFactory=DEBUG, output
log4j.logger.ee.sk.digidoc.factory.SAXDigiDocFactory=DEBUG, output
log4j.logger.ee.sk.digidoc.factory.PKCS11SignatureFactory=INFO, output
log4j.logger.ee.sk.xmlenc.factory.EncryptedDataSAXParser=INFO, output
log4j.logger.ee.sk.xmlenc.factory.EncryptedStreamSAXParser=INFO, output
log4j.logger.ee.sk.digidoc.DataFile=INFO, output
log4j.logger.ee.sk.xmlenc.EncryptedData=INFO, output
```

```
log4j.logger.ee.sk.xmlenc.EncryptedKey=INFO, output
log4j.logger.ee.sk.digidoc.Base64Util=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
```

Configuring software token usage

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

For using software tokens, 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, add the following parameters:

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

Please note that when verifying signatures that are created with the parameter value "KEY_USAGE_CHECK=false", an error message "Error: 39 - Signer's cert does not have non-repudiation bit set!" is produced.

3.6. 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.tsl – Classes modeling the ETSI TS 102 231 V3.1.1. Trust Service Status List types
- **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
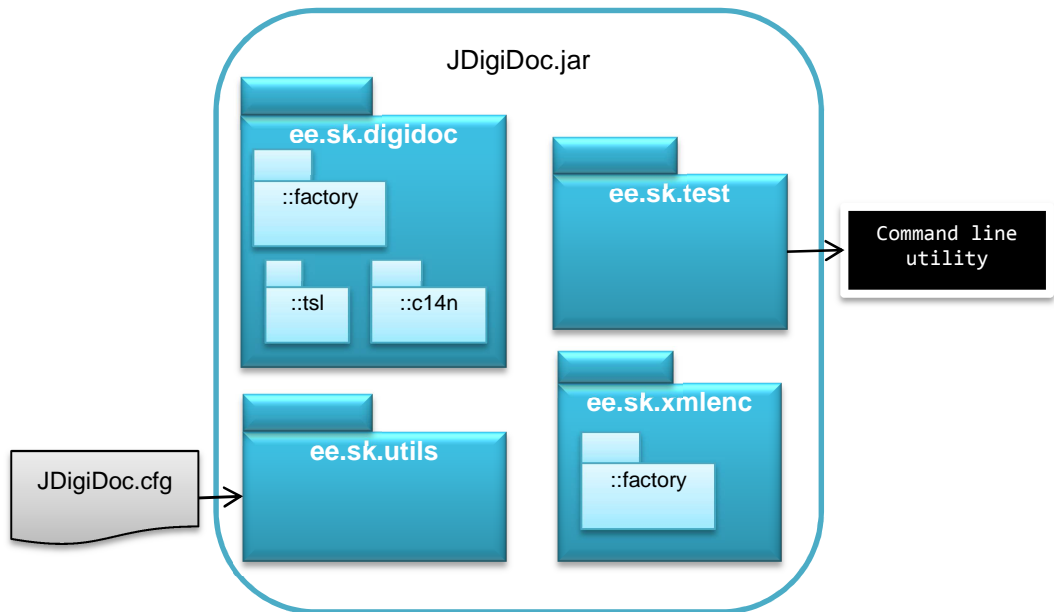
5 JDigiDoc packages

3.6.1. Package: ee.sk.digidoc
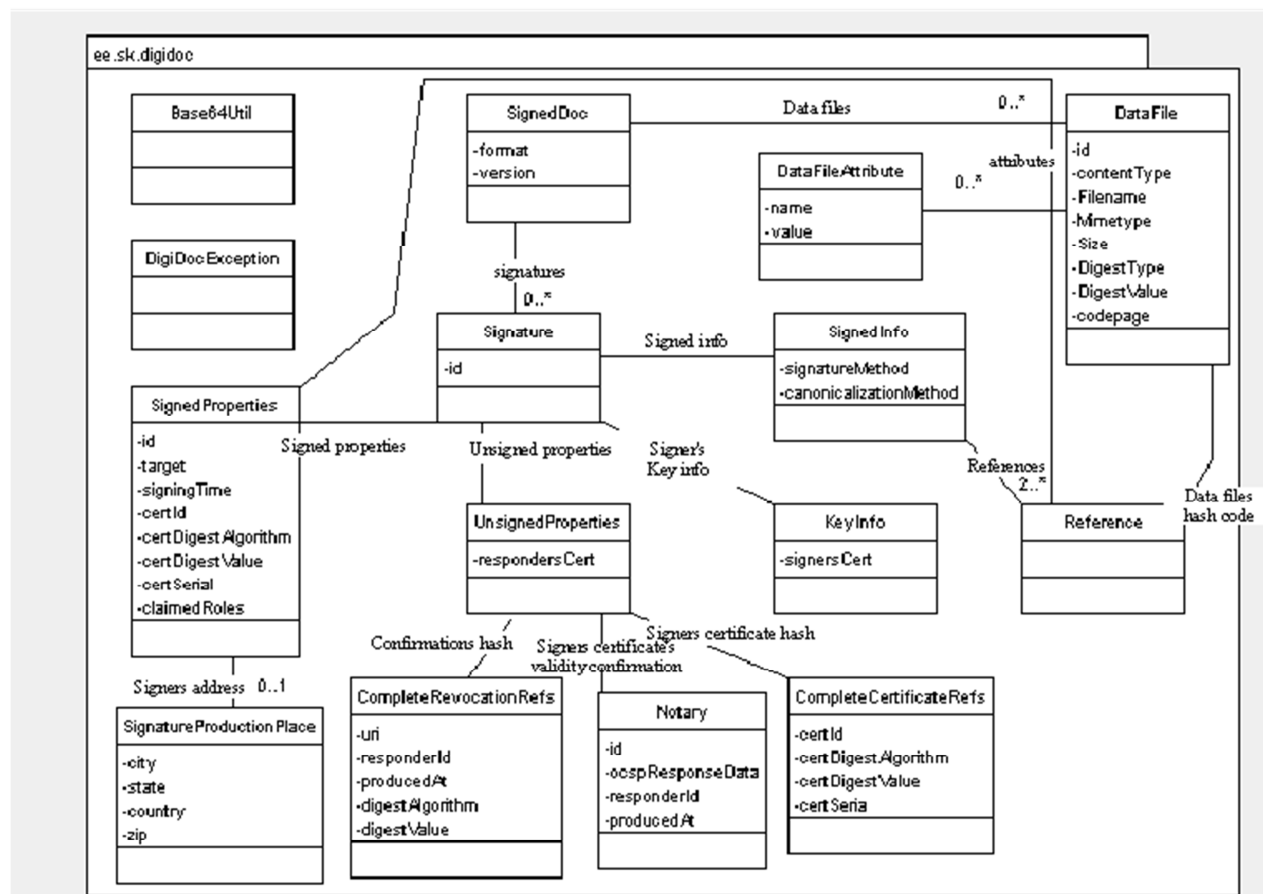
This package contains the core JDigiDoc classes modeling the structure of various XML-DSIG and XAdES entities.

- ee.sk.digidoc.SignedDoc – Models a digitally signed document in DIGIDOC-XML 1.3, 1.2 or 1.1 formats, in BDOCs case it models whole container. (DigiDoc)Document and (BDOC)Container can replace each other in this document depending which format is used. One document can contain one or more <DataFile> elements (payload data) and zero or more signatures (<Signature> element).

  Please use only the latest format version 1.3 for full XAdES conformity with DIGIDOC-XML.

- ee.sk.digidoc.DataFile – Payload data. Contains data from an external file that is being signed.

  DIGIDOC-XML uses either EMBEDDED_BASE64 mode (binary files; by default) or EMBEDDED mode (pure XML or text).

  Note: a previous third option - DETACHED mode (no actual data, just a reference to an external file and its hash code) - is no longer being supported for JDigiDoc.

  BDOC uses EMBEDDED mode.

- ee.sk.digidoc.DataFileAttribute – Optional user defined attributes for <DataFile> element. This way one can add some attributes to the actual data that are not part of the original file but also get signed. Required attributes are: Id, Filename, Contenttype, MimeType, Size. In case of using DETATCHED mode the following attributes are also required: DigestType, DigestValue, Codepage. User can add any attributes that don't collide with the names of required attributes.

- ee.sk.digidoc.Signature – Digital signature (XML-DSIG)

- ee.sk.digidoc.SignedInfo – actually signed content containing references and hash codes of other signed data objects (XML-DSIG).

- ee.sk.digidoc.Reference – a reference to a signed data object and its hash code (XML-DSIG).

- ee.sk.digidoc.KeyInfo – Signers certificate data (XML-DSIG). Please note that even if XML-DSIG allows various other possibilities of establishing the signer's identity like sending the signer's distinguished name (DN) etc., we still require the whole signer's certificate to be included in this element.

- ee.sk.digidoc.SignedProperties – additional signed properties (XAdES)

- ee.sk.digidoc.SignatureProductionPlace – optional signers address (XAdES).

- ee.sk.digidoc.UnsignedProperties – unsigned signature properties (XAdES)

- ee.sk.digidoc.Notary – OCSP confirmation

- ee.sk.digidoc.CompleteCertificateRefs – signer's certificate info and hash code (XAdES).

- ee.sk.digidoc.CompleteRevocationRefs – OCSP confirmation info and hash code (XAdES).

- ee.sk.digidoc.DigiDocException – specific Exception class for JDigiDoc exceptions.

- ee.sk.digidoc.Base64Util – Base64 decoder and encoder.

- ee.sk.digidoc.CertID – contains certificate info (not certificate itself). Such info is stored in XAdES documents in <SignersCertificate> and <Cert> elements. It contains certificates serial number, issuer DN and hash code.

- ee.sk.digidoc.CertValue – contains certificate itself. Certificates are added to XAdES signatures <CertificateValues> section and they are used when verifying the signature.

- ee.sk.digidoc.TimestampInfo – contains timestamp info. This version of JDigiDoc supports XAdES-X-L format with <SignatureTimeStamp> and <SigAndRefsTimeStamp>, but provides only verification of such documents.

3.6.2. Package: ee.sk.digidoc.factory

```
ee.sk.digidoc.factory

  DigiDocFactory          NotaryFactory            CanonicalizationFactory      SignatureFactory

  -init()                 -init()                  -init()                      -init()
  -readSignedDoc()        -getConfirmation()       -canonicalize()              -getAvailableTokenNames()
                          -parseAndVerifyResponse()                             -sign()
                          -getNotaryCert()                                      -getCertificate()
                                                                                -reset()

  SAXDigiDocFactory       IAIKNotaryFactory        DOMCanonicalizationFactory
                                                                                PKCS11SignatureFactory


  SAXDigiDocException
```

This package contains interfaces for exchangeable modules and their implementations.

- ee.sk.digidoc.factory.DigiDocFactory – interface for digidoc file parser

- ee.sk.digidoc.factory.SAXDigiDocFactory – implementation parsing digidoc documents using a SAX parser.

- ee.sk.digidoc.factory.SAXDigiDocException – DigiDoc specific parsing exception class that is a subclass of SAXException.

- ee.sk.digidoc.factory.NotaryFactory – interface for getting OCSP confirmations.

- ee.sk.digidoc.factory.IAIKNotaryFactory - Implementation module for getting OCSP confirmations. This module uses IAIK JCE library. You have to get a license of IAIK-JCE to use this.

- ee.sk.digidoc.factory.BouncyCastleNotaryFactory - Implementation module for getting OCSP confirmations. This module uses BouncyCastle library and requires no license.

- ee.sk.digidoc.factory.CanonicalizationFactory – interface for XML canonicalization.

- ee.sk.digidoc.factory.DOMCanonicalizationFactory – Implementation for XML canonicalization using the Apache XML Security library.

- ee.sk.digidoc.factory.SigantureFactory – Interface for signing.

- ee.sk.digidoc.factory.PKCS11SignatureFactory – Implementation for signature module using smartcards and a PKCS#11 driver. This module uses IAIK PKCS#11 wrapper library to access external native language PKCS#11 drivers. IAIK PKCS#11 wrapper is free ware.

- ee.sk.digidoc.factory.TimestampFactory – interface for timestamp operations.

- ee.sk.digidoc.factory.BouncyCastleTimestampFactory – implementation for timestamp operations using BouncyCastle library. In this version provides only verification of timestamps.

3.6.3. Package: ee.sk.utils
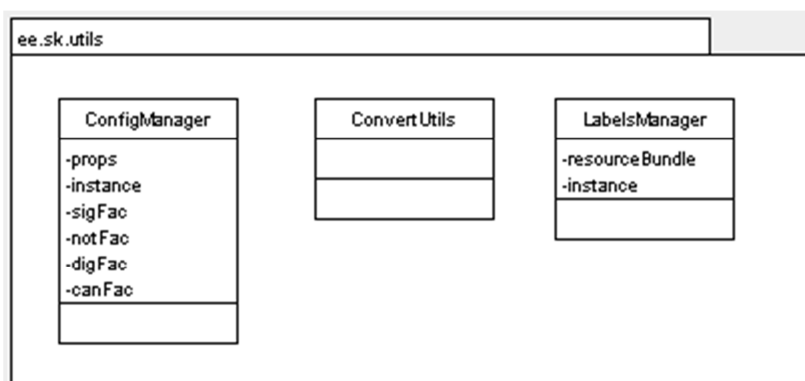
This package contains various utility classes.

- ee.sk.util.ConfigManager – configuration data utility.

- ee.sk.util.ConvertUtils – contains data conversion methods.

- ee.sk.util.LabelsManager – manages labels for GUI, not used at the moment.

3.6.4. Package: ee.sk.test

This package contains sample programs

- ee.sk.test.Test1 – creates a digidoc document in format: DIGIDOC-XML, version: 1.3 adds three data files, one in each possible mode, signs the document using Estonian ID card, card reader and a PKCS#11 driver, gets an OCSP confirmation to this signature, writes everything in a digidoc file, reads the file in again and verifies signatures and OCSP confirmations.

- ee.sk.test.Test2 – Reads in a digidoc document and verifies signatures and OCSP confirmations.

- ee.sk.test.jdigidoc – command line utility program that allows one to invoke most of the functionality offered by the library.

3.6.5. Package: ee.sk.xmlenc

This package contains classes modeling the XML entities specified in XML-ENC standard.

- ee.sk.xmlenc.EncryptedData – Corresponds to the <EncryptedData> element in XML-ENC standard. JDigiDoc doesn't support many encrypted data objects or a mix of encrypted and unencrypted data in one xml document. One encrypted document can have only one <EncryptedData> element, which is also the documents root element and it contains one <EncryptedKey> element for every recipient (e.g. Possible decrypter) of the document and a set of <EncryptionProperty> elements to store any metadata.

- ee.sk.xmlenc.EncryptedKey – Corresponds to the <EncryptedKey> element of XML-ENC standard. One encrypted document contains on <EncryptedKey> element per every recipient that contains the 128 bit AES transport key encrypted by the recipients certificate and any other data used to identify the key.

- ee.sk.xmlenc.EncryptionProperties - Corresponds to the <EncryptionProperties> element of XML-ENC standard. It contains one or more EncryptionPropety objects.

- ee.sk.xmlenc.EncryptionProperty - Corresponds to the <EncryptionProperty> element of XML-ENC standard. XML-ENC standard defines optional attributes "Id" and "Target" and allows to add any custom attributes. JDigiDoc uses the attribute "Name" to identify special elements that the library uses to store metadata.

3.6.6. Package: ee.sk.xmlenc.factory

This package contains parsers of encrypted files.

- ee.sk.xmlenc.factory.EncryptedDataParser – defines a general interface for parsing encrypted files. Suitable for only smaller files because collects all data in memory

- ee.sk.xmlenc.factory.EncryptedDataSAXParser – implements the EncryptedDataParser interface with a SAX parser.

- ee.sk.xmlenc.factory.EncryptedStreamParser – defines an interface for decrypting encrypted files. Suitable for larger files but doesn't keep any document info in memory. Decrypts data directly from input stream and writes to output stream.

- ee.sk.xmlenc.factory.EncryptedStreamSAXParser- implements the EncryptedStreamParser interface with a SAX parser.

3.7.   Digital signing

JDigiDoc library offers creating, signing and verification of digitally signed documents, according to XAdES (ETSI TS101903) and XML-DSIG standards. In 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.

3.7.1. 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".

3.7.2. 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);

//format - SK-XML, DIGIDOC-XML, BDOC
//version - 1.0, 1.1, 1.2, 1.3, bdoc has only 1.0
```

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

Note that you can use also the older format SK-XML and format versions 1.1 and 1.2 for DIGIDOC-XML, but those are not fully XAdES compliant and are maintained only for historical reasons and backward compatibility.

In case of BDOC, set the profile value for the created SignedDoc object. For example, you can use the default profile value that is defined by parameter "DIGIDOC_DEFAULT_PROFILE" in configuration file:

```
sdoc.setProfile(ConfigManager.instance().getStringProperty("DIGIDOC_DEFAULT_PROFILE", 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.

3.7.3. Adding data files

Payload data can be added in two modes:

- EMBEDDED_BASE64 – use this to embed binary data in base64 encoding:

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

- EMBEDDED – use this to embed pure text or XML:

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

- DETATCHED – use this to add a reference to an external file (file to big etc.).

    Note: the DETATCHED mode is no longer considered supported for JDigiDoc.

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

The new objects are created in memory but the references 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)

If you don't want JDigiDoc to read the files because you hold the data in a database, generate it dynamically etc., then read it yourself and assign it to the DataFile object using the setBody() method. Once you have assigned the data to this object it will no longer be read from disk:

```
String myBody = "My sample data with umlauts äüõ";
df.setBody(myBody.getBytes("ISO-8859-1"), "ISO-8859-1");
```

It is your task to know what code page is used by your data. In this example we use the ISO-8859-1 code page.

Now JDigiDoc stores the original code page in the Codepage="ISO-8859-1" attribute of <DataFile> to be able to convert the data back to original code page later.

All data in digidoc documents is kept in UTF-8, so we had to convert the data in the previous sample.

If your data is already in UTF-8 the do the following:

```
byte[] u8b = ConvertUtils.str2data(myBody);
df.setBody(u8b, "UTF-8");
```

If the DataFile contains pure XML or text then you can get the data using:

```
String body = df.getBodyAsString();
```

3.7.4. Adding signatures

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

- using an Estonian ID card or

- any other smartcard provided that 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 (web-application?) and then add the signature value to digidoc document.

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>";

// Do card login and get certificate

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

//Index of the key pair used for signing. In this sample we use 0
//as it's used on Estonian ID cards. Please note that this index
//starts with 0 and counts ONLY the key pairs usable for digital
//signature. E.g. no authentication key pairs!

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

Now compute the payload data hash codes and create a Signature object:

```
Signature sig = sdoc.prepareSignature(cert,
        null, // String[] roles
        null); // SignatureProductionPlace (address)
```

The signature is not complete yet as it's missing the actual RSA signature data, but we have now calculated the final 20 byte SHA1 hash that serves as input for RSA signature.

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

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

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

Now add signature value to the Signature object.

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

3.7.5. Adding an OCSP confirmation

Call the following method to add OCSP confirmation:

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

After adding an OCSP confirmation, the signature is now complete and provides long-time proof of the signed data.

If you just want to verify the validity of a certificate by using OCSP, not to sign a document (for example when authentication users to your application) then do:

```
public void NotaryFactory.checkCertificate(X509Certificate cert)
        throws DigiDocException;
```

3.7.6. Reading and writing digidoc documents

Write a SignedDoc object to a digidoc file as follows:

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

Read a digidoc document as follows:

```
DigiDocFactory digFac = ConfigManager.instance().getDigiDocFactory();
SignedDoc sdoc = digFac.readSignedDoc("<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);
```

3.7.7. Verifying signatures and OCSP confirmations

After having read a digidoc document verify the signatures as follows:

```
ArrayList errs = null;
SignedDoc.verify(checkDate, // Should always be "true"
        demandConfirmation); // In case of ddoc, always "true".
        // In case of bdoc, depends on profile.
```

This method verifies all signatures one by one. If the signature has an OCSP confirmation then this too is being verified.

In case of signature verification 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. If the returned container was empty, then the document verified ok.

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.

3.7.8. Validating digidoc documents

For validating the structure of a digidoc document, JDigiDoc classes contain methods that validate the contents of their 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.

Use the method:

```
ArrayList SignedDoc.validate(boolean bStrong);
```

This method returns an array of DigiDocException objects which could be displayed in some user interface. If the array is empty then the document is ok.

Verifying signatures calls also this method, but using `bStrong=false`, so it might still accept some smaller errors if this doesn't invalidate the signatures.

3.8. Encryption and decryption

In addition to digital signing JDigiDoc library offers also digital encryption and decryption according to the 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.

JDigiDoc additionally enables to compress the data with ZLIB algorithm before encryption. It encrypts data 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.

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)

- 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.

JDigiDoc doesn't support many encrypted data objects or a mix of encrypted and unencrypted data in one XML document.

One encrypted document:

- contains only one <EncryptedData> element, which is also the documents 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.

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

3.8.1. Composing encrypted documents

In order to compose an encrypted document you have to:

- create the EncryptedData object first,

- add all recipients' certificates and other data,

- add the unencrypted data, encrypt and possibly compress it

- finally store it in a file or other medium.

```
EncryptedData cdoc = new EncryptedData(
      null, // optional Id attribute value
      null, // optional Type attribute value
      null, // optional Mime type 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.

If encrypting a digidoc document, you should assign the "Type" 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`

If encrypting pure XML documents then one could assign to attribute "MimeType" the value:

- `"text/xml"`

which has also been defined as a constant:

- `EncryptedData.DENC_ENCDATA_MIME_XML`

JDigiDoc library uses the attribute "MimeType" also to store the fact that the data has been packed with ZLIB algorithm before encryption. The library assigns to MimeType attribute the value

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

which has also been defined as a constant:

- `EncryptedData.DENC_ENCDATA_MIME_ZLIB`

You don't have to do this yourself. JDigiDoc assigns this value when packing the data and if the MimeType attribute was not empty before that then the old value is stored in an <EncryptionProperty Name="OriginalMimeType"> subelement.

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

3.8.2. Adding recipient info

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

For every recipient the library stores:

- the AES transport key encrypted with the recipients certificate

- the certificate itself

- possibly some other data used to identify the key.

A certificate that is usable for data encryption must be used. In case of Estonian ID cards it's the authentication certificate.

For adding an EncryptedKey object the recipient's certificate must be in PEM format.

EncryptedKey object is 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 ee.sk.test.jdigidoc assigns a unique value to every EncryptedKey objects "Recipient" attribute. It could be the recipients forename or something more complicate like "<last-name>,<first-name>,<personal-code>". This is later used as a command line option to identify the recipient who's 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 applications proper functioning. Something from the certificate like its CN attribute should be used to identify the recipient.

3.8.3. Encryption and data storage

There are two possible methods for encrypting data:

- Small data objects – does all operations in memory.

  Faster and more flexible but requires more memory. The compression option "BEST EFFORT" can be used which means that data will be compressed and if this resulted in reduction of data size then it's used, otherwise it will be discarded and original data is encrypted uncompressed.

- Big data objects – reads and handles all data in blocks of fixed size.

Capable of encrypting large sets of data but less flexible. Compression or no compression must be specified for this operation. It doesn't offer encrypting in memory, so input and output streams must be provided. Note that the functionality of encrypting big data sets is not currently tested.

For encrypting small data objects:

- create the EncryptedData object first

- add all recipient info, the unencrypted data

- encrypt it, possibly compressing the data

- store it in a file or another medium.

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);

// store the original filename and or mime type if applicable

cdoc.addProperty(EncryptedData.ENCPROP_FILENAME, inFile);

// Encryption. Options:
//EncryptedData.DENC_COMPRESS_ALLWAYS,
// EncryptedData.DENC_COMPRESS_NEVER and
EncryptedData.DENC_COMPRESS_BEST_EFFORT

cdoc.encrypt(EncryptedData.DENC_COMPRESS_BEST_EFFORT);
FileOutputStream fos = new FileOutputStream(outFile);
fos.write(m_cdoc.toXML());
fos.close();
```

For encrypting bigger data sets:

- create the EncryptedData object first

- register all recipients, add any metadata

- encrypt the data by reading input stream, possibly compressing the data and writing to output stream.

```
// store metadata such as the original file name.

cdoc.addProperty(EncryptedData.ENCPROP_FILENAME, inFile);
// Encryp. Compression options are only
EncryptedData.DENC_COMPRESS_ALLWAYS

// and EncryptedData.DENC_COMPRESS_NEVER

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

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.

3.8.4. Parsing and decrypting

There are also two options 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.

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. A way to identify the EncryptedKey object to use for decryption has to be provided. The current implementation uses the "Recipient" attribute for this purpose. Note that the functionality of decrypting big data sets is not currently tested.

1. 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:

```
m_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(m_cdoc.getData());
fos.close();
```

2. For decrypting big encrypted documents use:

```
// provide input and outputstreams

FileInputStream fis = new FileInputStream(inFile);
FileOutputStream fos = new FileOutputStream(outFile);
EncryptedStreamParser streamParser = ConfigManager.
     instance().getEncryptedStreamParser();

// decrypt and possibly decompress the data

streamParser.decryptStreamUsingRecipientName(fis, fos,
     0,  // smartcards Tokeni index. For Estonian ID cards always 0
     pin, // smartcards PIN code. For Estonian ID card PIN1
     recvName); // selected EncryptedKey objects Recipient attribute
fos.close();
fis.close();
```

Data is read from input stream, decrypted, possibly decompressed and written to output stream.

4. JDigiDoc utility

The command line utility program - ee.sk.test.jdigidoc - can be used to test the library or simply use it directly to encrypt, decrypt and digitally sign documents.

1 Using the command line utility program with Windows Prompt: ee.sk.test.jdigidoc

The general format is:

```
> java ee.sk.test.jdigidoc [commands]
```

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

```
> java ee.sk.test.jdigidoc -help
```

4.1. Prerequisites

For running the program from the command line, the appropriate class path must be set first for finding all the necessary classes and other resource files.

Examples for setting the class path:

- by using the –classpath option with specifying all the necessary class path values when calling a JDK tool

```
> java -classpath classpath1;classpath2...
```

- by setting the CLASSPATH environment variable:

```
> set CLASSPATH=classpath1;classpath2...
```

- by setting the class path variable in a batch file and using the –classpath option when calling the JDK tool

```
> jdigidoc.bat //for setting the %CP% variable value
> java –classpath %CP% ee.sk.test.jdigidoc –ddoc-new…
```

- by setting the class path value and the program execution command in a batch file and using the –classpath option with up to 9 additional arguments from the command line

```
> jdigidoc.bat -ddoc-new…
```

A sample batch file can contain the following entries:

1. **JDIGIDOC_HOME** variable for setting the folder containing the JDigiDoc libraries, e.g.:

```
REM setting JDIGIDOC_HOME variable to the correct folder
set JDIGIDOC_HOME=C:\Temp\JDigiDoc
```

2. **CP** variable for setting paths to all the external libraries and your certificate files used, e.g.:

```
REM setting the CP variable to correct class search paths
REM depending on your java version
REM enter correct version of bcprov*.jar from lib folder
set CP=%JDIGIDOC_HOME%\lib\bcprov-jdk16-145.jar;%JDIGIDOC_HOME%\lib\bcmail-
jdk16-145.jar;%JDIGIDOC_HOME%\lib\bcpg-jdk16-
145.jar;%JDIGIDOC_HOME%\lib\bctsp-jdk16-
145.jar;%JDIGIDOC_HOME%\lib\iaik_jce.jar;%JDIGIDOC_HOME%\lib\iaikPkcs11Wrapp
er.jar;%JDIGIDOC_HOME%\lib\jakarta-log4j-
1.2.6.jar;%JDIGIDOC_HOME%\lib\xmlsec.jar;%JDIGIDOC_HOME%\lib\xalan.jar;%JDIG
IDOC_HOME%\lib\xercesImpl.jar;%JDIGIDOC_HOME%\lib\xml-
apis.jar;%JDIGIDOC_HOME%\lib\xmlParserAPIs.jar;%JDIGIDOC_HOME%\lib\ant.jar;%
JDIGIDOC_HOME%\lib\mail-1.4.jar; %JDIGIDOC_HOME%\lib\jdcerts.jar;
```

3. **Optional**: **ee.sk.test.jdigidoc** for executing the program, e.g.:

```
REM – optional command to run the program –
REM - 1- C:\...\bin\java sets your java bin folder
REM - 2- Xmx512m is the heap size for Java virtual machine
REM – 3- classpath %CP% sets the classpath according to above
REM - 4. ee.sk.test.jdigidoc executes the program
REM – 5 -config sets default config file according to above
REM '%1 %2 %3 …' are placeholders to add arguments from user
C:\ jdk1.7.0\bin\java -Xmx512m -classpath %CP% ee.sk.test.jdigidoc -config
%JDIGIDOC_HOME%\jdigidoc-win.cfg %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Note: using this option in the batch file will only allow up to 9 additional arguments from the user to be passed on to the program from the command line, i.e.:

```
 > jdigidoc.bat -ddoc-new (%1) –ddoc-add (%2) test1.txt (%3) text/plain
(%4)….
```

4.2. General commands

- **-? 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:

- o   GOOD – certificate is valid

- o   REVOKED – certificate has been revoked

- o   UNKNOWN – certificate has never been issued or issuer is unknown

- o   EXPIRED – certificate has been expired

- o   SUSPENDED – certificate has been suspended

- o   OCSP_UNAUTHORIZED – if no access to OCSP validity confirmation service

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

> java ee.sk.test.jdigidoc –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
```

4.3.   Digital signature commands

- -**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-new –** creates a new DigiDoc container

- **-ddoc-add <input-file> <mime-type> –** adds a data file to a DigiDoc container

- **-ddoc-sign <pin-code> –** 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-list** - displays a DigiDoc file's content info and verifies signature(s)

- **-ddoc-validate -** displays and verifies a DigiDoc file's signature(s)

- **-ddoc-notarize** – adds an OCSP confirmation to a DigiDoc file. In case of BDOC file, timestamp value or OCSP confirmation is added according to profile.

- **-ddoc-extract <data-file-id> <output-file> -** extracts DigiDoc file's content

- **-mid-sign <phone-no> <per-code> [[<country>(EE)] [<lang>(EST)] [<service>(Testing)] [<manifest>] [<city> <state> <zip>]] –** signs a DigiDoc file using Mobile-ID

Creating new DigiDoc files

-ddoc-new [format] [version/profile]

Creates a new digidoc container in the specified format and version. The current default format is DIGIDOC-XML, version 1.3 (newest).

You can create new documents with JDigiDoc either in DIGIDOC-XML or BDOC **format**.

Note: the older SK-XML format is supported only for backward compatibility.

By using the optional parameter - version/profile – with this command, you can also specify:

- the format **version** to be created if using DIGIDOC-XML format

- or the signature's **profile** if using BDOC format

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

Adds a new data file to a digidoc document. If digidoc doesn't exist then creates one in the default format.

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).

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 or EMBEDDED_BASE64 (used by default).

-ddoc-sign <pin-code> [manifest] [country] [state] [city] [zip] [slot] [profile]

Adds a digital signature to the digidoc document. You can use it with following parameters:

pin code	Required.
	In case of Estonian ID cards, pin code2 is used for digital signing.
	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") and instead of the pin-code, the password for the PKCS#12 file container should be entered here.
manifest	Role or resolution of the signer
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 private key's slot on a smartcard.
	When operating for example with a single Estonian ID card, its signature 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.
	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.

-mid-sign <phone-no> <per-code> [[<country>(EE)] [<lang>(EST)] [<service>(Testing)] [<manifest>] [<city> <state> <zip>]]

Invokes mobile signing of a DDOC/BDOC file using Mobile-ID and DigiDocService.

Mobile-ID is a service based on Wireless PKI providing for mobile authentication and digital signing, currently supported by all Estonian and some Lithuanian mobile operators.

The Mobile-ID user gets a special SIM card with private keys on it. Hash to be signed is sent over the GSM network to the phone and the user shall enter PIN code to sign. The signed result is sent back over the air.

DigiDocService is a SOAP-based web service, access to the service is IP-based and requires a written contract with provider of DigiDocService.

You can use Mobile-ID signing with following parameters:

phone-no	Required. Phone number of the signer with the country code in format +xxxxxxxxx (for example +3706234566)
per-code	Required. Identification number of the signer (personal national ID number).
country	Country of origin. ISO 3166-type 2-character country codes are used (e.g. default is **EE**)
lang	Language for user dialog in mobile phone. 3-character capitalized acronyms are used. (e.g. default is **EST**)
service	Name of the service – previously agreed with Application Provider and DigiDocService operator. Maximum length – 20 chars. (e.g. default is **Testing**)
manifest	Role or resolution of the signer
city	City where the signature is created
state	State or province where the signature is created
zip	Postal code of the place where the signature is created

-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 ee.sk.test.jdigidoc -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 ee.sk.test.jdigidoc -ddoc-new BDOC -ddoc-add c:\temp\test1.txt text/plain –ddoc-sign 12345 -ddoc-out c:\temp\test1.bdoc

Input:
  - 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: creating new DigiDoc file with signing, with BDOC format and specified profile (BDOC_PROFILE_CL)

> java ee.sk.test.jdigidoc -ddoc-new BDOC CL -ddoc-add c:\temp\test1.txt text/plain –ddoc-sign 12345 -ddoc-out c:\temp\test1.bdoc

Input:
  - 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 ee.sk.test.jdigidoc -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: using Mobile-ID for signing
> java ee.sk.test.jdigidoc -ddoc-new BDOC -ddoc-add c:\temp\test1.txt text/plain –mid-sign +3706234566 41110170240 -ddoc-out c:\temp\test1.bdoc

Input:
  - c:\temp\test1.txt   - a data file to be added to container
  - text/plain   - mime type of the data file
  - +3706234566   - signer's mobile number
  - 41110170240   - signer's personal code
  - c:\temp\test1.bdoc - container to be created

Sample: signing an existing digidoc container with a technical signature
> java ee.sk.test.jdigidoc -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
(manifest, country, state, city, zip)
 - 0                   - slot
 - "BES"               - profile identifier of a technical signature
 - c:\temp\test1.ddoc  - output (modified) digidoc container


Sample: Adding multiple data files to an existing unsigned DigiDoc container
> java ee.sk.test.jdigidoc -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
```

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-list

Displays the data file and signature info of a digidoc just read in; verifies all signatures. Returns:

- o **Digidoc container data**, in format

  DigiDoc document: <format identifier> <version> (e.g. DIGIDOC-XML 1.3)

  profile: <signature profile> (e.g. TM)

- o **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)

- o **List of all signatures** (if existing), in format:

  Signature: S0 profile: <signature profile> (e.g. TM)

  Signature: S0 profile: <signature profile> key: < signer's key info validation results> (e.g. TM key: --> OK)

  Signature: S0 profile: <signature profile> - <signer's key info: personal code, first name, last name> <validation results> (e.g. TM - 41109140240,MARI-LIIS,MÄNNIK --> OK)

-ddoc-validate

Validates the DigiDoc file just read in.

Returns the DigiDoc document's **validation results**:

- o "Validation --> OK" or "Validation --> ERROR".

Additionally, if validating a signed document, the signatures are verified after the document validation.

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

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

-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: listing DigiDoc file's contents, not signed
> java ee.sk.test.jdigidoc -ddoc-in c:\temp\test1.ddoc -ddoc-list

Input:
- c:\temp\test1.ddoc – the digidoc file which contents are to be listed

Returns:
DigiDoc document: DIGIDOC-XML/1.3 profile: TM
        DataFile: D0 file: test2.txt mime: text/plain size: 8
        DataFile: D1 file: test1.txt mime: text/plain size: 8

Sample: listing DigiDoc file's contents, signed
> java ee.sk.test.jdigidoc -ddoc-in c:\Temp\test2.ddoc -ddoc-list

Input:
  - c:\temp\test2.ddoc – the digidoc file to be validated/contents listed

Returns:
DigiDoc document: DIGIDOC-XML/1.3 profile: TM
  DataFile: D0 file: test2.txt mime: text/plain size: 8
  Signature: S0 profile: TM
  Signature: S0 profile: TM key: OK
  Signature: S0 profile: TM - 41109140240,MARI-LIIS,MÄNNIK --> OK

Sample: validating existing DigiDoc file, signed
> java ee.sk.test.jdigidoc -ddoc-in c:\Temp\test2.ddoc -ddoc-validate
Input:
- C:\temp\test2.ddoc   – the digidoc file to be validated
Returns:
Validation --> OK
Signature: S0 profile: TM - 41109140240,MARI-LIIS,MÄNNIK --> OK

Sample: Extracting a data file from an existing DigiDoc file
> java ee.sk.test.jdigidoc -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
```

4.4. Encryption commands

- **-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

- **-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 DigiDoc software components, 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 other DigiDoc software components, 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> <recipient> <pin> <output-file>* - used for decrypting large files in any original format
   - *-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

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 ee.sk.test.jdigidoc -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
```

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.

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 ee.sk.test.jdigidoc -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 ee.sk.test.jdigidoc -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 ee.sk.test.jdigidoc -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 ee.sk.test.jdigidoc –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 ee.sk.test.jdigidoc -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
```

Decrypting files

-cdoc-decrypt-sk <pin> <output-file>

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.

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>

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.

-cdoc-decrypt-stream <input-file> <recipient> <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).

Recipient (required) represents recipient's attribute of the encrypted document**.** This value depends on the recipient attribute set during encryption when using the –cdoc-recipient command. It is recommended to pass on the entire CN field value from the recipient's certificate as the recipient attribute there (e.g. if CN=MÄNNIK,MARI-LIIS,41110210240, then recipient = MÄNNIK,MARI-LIIS,41110210240)

Otherwise, if no recipient attribute has been specified during encryption, then only the first part of the certificates CN value is used – e.g. if the certificate's CN=MÄNNIK,MARI-LIIS,41110210240, then recipient=MÄNNIK.

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 ee.sk.test.jdigidoc -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 ee.sk.test.jdigidoc -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 ee.sk.test.jdigidoc -cdoc-decrypt-stream c:\Temp\test1_large.cdoc
 MÄNNIK,MARI-LIIS,41110212444 1234 c:\Temp\test1_d.ddoc

Input:
  - c:\Temp\test1_large.cdoc   – the encrypted file to be decrypted
- MÄNNIK,MARI-LIIS,41110212444 - the recipient's ID (= certificate's CN)
  - 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 ee.sk.test.jdigidoc -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
```

5. National and cross-border support

5.1. National PKI solutions and support

5.1.1. Supported Estonian Identity tokens

The Digital Signature Act (DSA), passed by the Estonian parliament in 2000, forms the legal framework for digital signatures in Estonia, equating advanced electronic signatures (in terms of EC directive 1999/93/EC) to handwritten ones, as long as they are compliant with the DSA's requirements.

ID cards

Since 2002, Estonia has issued PKI-enabled ID cards to over 90% of its citizens and permanent residents. The card has been integrated into a national public-key infrastructure and is mandatory for citizens over the age of 15.

Upon the initialization of a new ID card for the user, two RSA key pairs are loaded into it. Certificates binding the public keys to the user are also issued and stored on the card as well as in a public database. The certificates are issued by a certification authority in the list of state-recognized CAs - **AS Sertifitseerimiskeskus (SK).** The intended uses for the private keys, protected by two separate PIN codes, are identification (for the first key pair) and signing (for the second key pair). The certificates contain the holder's name and personal code (national ID code). In addition, the authentication certificate contains the holder's unique e-mail address. Certificates on the ID-card are "Qualified" in terms of EC 1999/93.

Mobile-IDs

Since 2007, EMT (the largest Estonian mobile operator) in cooperation with SK has issued also mobile SIM cards with similar functionality as ID cards (user authentication and digital signing). Since 01.02.2011, the Mobile-ID is considered an official digital identification document in Estonia. Similarly, RSA key pairs are loaded into those cards and the public keys are issued certificates binding them with users. Corresponding certificates are also qualified ones thus serving alternative option to smartcard-based PKI. This project currently covers all Estonian mobile operators (EMT, Elisa, Tele2) and also Lithuanian mobile operator Omnitel and is opened to other providers in the Baltic region.

Organizational certificates (Digital stamps)

Additionally, SK issues certificates to organizations and private companies that can be used to sign documents digitally. These are technically equal to personal signing certificates and their legal use is also regulated by the DSA in Estonia.

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

Token	Type	Description	Supported JDigiDoc functionality
EstEID 3.0 and 1.0	Certificate –based PKI smart cards	Different Estonian ID card versions issued between: • 2002 – 2011 • From 01.01.2011 onwards (using new chip platform) • From 10.07.2011 onwards (certificates issued by new root - EECCRCA)	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
Mobile-ID	PKI capable SIM-card	Carrier for Mobile-IDs in Estonia, issued by mobile phone operators (EMT, Elisa, Tele2)	Signing
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.

5.1.2. 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 the service:
  - o having a contract with SK and accessing the service from a specific IP address(es) – as practiced **by companies/services**
  - o 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)

Supported CA 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
- added to Trusted Service List as 'local' CA certificates by default
- registered in the JDigiDoc configuration file.

For adding new 'local' CAs, OCSP responders or certificates, please refer to Section 3.5, **Configuring JDigiDoc,** under **CA certificates** and **OCSP responder certificates.**

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
		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 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 2010 RESPONDER	*26-Aug- 2010*	EID-SK 2007 OCSP Responder
	KLASS-SK		05-May-2012	for organizational certificates issued until 10.2010
		KLASS-SK OCSP RESPONDER	*05-Apr- 2006*	KLASS-SK OCSP Responder
		KLASS-SK OCSP 2006 RESPONDER	*27-Mar-2009*	KLASS-SK OCSP Responder
		KLASS-SK OCSP 2009 RESPONDER	*04-May- 2012*	KLASS-SK OCSP Responder
	KLASS-SK 2010		26-Aug-2016	for organizational certificates issued from 10.2010
		KLASS-SK 2010 OCSP RESPONDER	*26-Aug- 2016*	KLASS-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
		SK OCSP 2011 RESPONDER	*18-Mar- 2024*	common OCSP responder for all certificates issued under EECCRCA
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 2005	*07-Apr-2012*	TEST-SK OCSP

		RESPONDER		responder
TEST EECCRCA			18-Dec-2030	SK's 2ⁿᵈ test root certificate
	TEST ESTEID-SK 2011		07-Sep-2023	for test ID cards, Digi-ID and Mobile-ID certificates issued from 04.2011
	TEST EID-SK 2011		07-Sep-2023	for all other test certificates issued from 04.2011
		Test SK OCSP 2011 RESPONDER	*07-Sep-2024*	*common OCSP responder for all test certificates issued under TEST-EECCRCA*

All of the above listed SK certificates are also downloadable from www.sk.ee/certs.

5.1.3. Supported Lithuanian Identity tokens

The Lithuanian Electronic Signature Law (ESL) forms the legal framework for digital signatures in Lithuania, according to which electronic signatures will have the same legal effect as any hand-written signature.

ID cards and USB tokens

The Lithuanian personal identity cards have been issued since 1 January 2009 and are mandatory for citizens over the age of 16. The ID card is a PKI-based smart card and incorporates two certificates: one for authentication, and one for electronic signatures, with only the latter being considered as qualified. The Lithuanian national CA - **Residents' Register Service (NSC)** - under the Lithuanian Ministry of Internal Affairs is responsible for issuing the ID card certificates.

Additionally, the CA **Centre of Registers Certificate Center (RCSC)** is a supplier of qualified certificates in Lithuania and offers digital certification services using various cryptographic electronic signature mediums (smart cards, USB tokens, SIM cards) to Lithuanian residents based on their ID document (passport or identity card).

Mobile-IDs

Since November 2007, it has been possible to sign documents electronically using a mobile phone with a SIM card which offers similar authentication and digital signing functionality as the ID card and consists of qualified certificates to identify the personality of the user and to sign documents. This project currently covers the Lithuanian mobile operator Omnitel.

Token	Type	Description	Supported JDigiDoc functionality
LTU ID	Certificate –based PKI smart card	Lithuanian ID cards issued since 2009	All JDigiDoc functionalities
Aladdin eToken Pro	Certificate –based PKI USB authenticator	Carrier for certificates issued to Lithuanian residents by RCSC, for authentication and qualified digital signature creation	All JDigiDoc functionalities
Mobile-ID	PKI capable SIM-card	Carrier for Mobile-IDs in Lithuania, issued by mobile phone operators (Omnitel)	Signing

5.1.4. Trusted Lithuanian Certificate Authorities

The currently supported certification authorities (CAs) issuing qualified certificates in Lithuania are the **Residents' Register Service (NSC**, www.nsc.vrm.lt) and **Centre of Registers,** operating the **Certification Centre** (**RCSC**, http://www.registrucentras.lt/rcsc/index_en.php)

Additionally, certain Lithuanian commercial entities also use qualified certificates issued by the Estonian CA AS Sertifitseerimiskeskus (SK).

Supported CA hierarchy chains

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

- included in the JDigiDoc distribution

- added to Trusted Service List as 'local' CA certificates by default

- registered in the JDigiDoc configuration file.

NSC's hierarchy chains:

Certificate Common Name (CN)		Valid to	Description
Nacionalinis sertifikavimo centras (RootCA)		05-Nov-2026	NSC's 1st root certificate
	Nacionalinis sertifikavimo centras (PolicyCA)	06-Nov- 2014	
	Nacionalinis sertifikavimo centras (IssuingCA)	06-Nov- 2014	
Nacionalinis sertifikavimo centras (RootCA)		12-Jun-2027	NSC's 2nd root certificate
	Nacionalinis sertifikavimo centras (PolicyCA)	12-Jun-2021	
	Nacionalinis sertifikavimo centras (IssuingCA)	13-Jun- 2015	

RCSC's hierarchy chains:

Certificate Common Name (CN)		Valid to	Description
VI Registru Centras RCSC (RootCA)		21-Jul-2024	RCSC's 1st root certificate
	VI Registru Centras RCSC (PolicyCA)	24-Sep- 2016	
	VI Registru Centras RCSC (IssuingCA)	24-Sep 2012	

Note: For RCSC, their entire CA hierarchy chain's certificates are available from http://www.registrucentras.lt/bylos/rcsc/root_certificates.zip. For using additional RCSC certificates with JDigiDoc which are not listed in the above table, you need to download them to your file system and add corresponding entries to your configuration file as follows:.

For changing the CA's and certificate settings in JDigiDoc, there are two main options:

- you can set the DIGIDOC_TLS_DIR parameter in the configuration file to use TSL files (in XML format) placed in a specified folder where all the trusted CAs and their OCSP responder data can be retrieved from:

```
DIGIDOC_TSL_DIR =        ./jdigidoc/src/main/tsl
```

- you can register new 'local' CA's, OCSP responders and certificates directly in the configuration file. For example, for adding a new CA, the following parameters should be updated:

```
DIGIDOC_USE_LOCAL_TSL   =       TRUE <must be set to true for using also the local
CA's registered below>

DIGIDOC_CAS     =       <add +1 if adding a new CA to the ones already registered>
…
DIGIDOC_CA_4_NAME       =       <enter name for a new CA, e.g. TEST4>
DIGIDOC_CA_4_TRADENAME  =       <enter short name for a new CA, e.g. T4>
DIGIDOC_CA_4_CERTS      =       <update or enter the number of CA certs, e.g. 3>
DIGIDOC_CA_4_CERT1      =       <enter locations for all the CA certs>
DIGIDOC_CA_4_CERT2      =       <e.g. jar://certs/CA_4_TESTcert2.crt>
DIGIDOC_CA_4_CERT3      =       <e.g. jar://certs/CA_4_TESTcert3.crt>
…
DIGIDOC_CA_4_OCSPS = <update or enter the number of OCSP responders for a CA, e.g. 1>
…
DIGIDOC_CA_4_OCSP1_CA_CN =<enter common name for CA, e.g. TEST4>
DIGIDOC_CA_4_OCSP1_CA_CERT =<enter cert for CA, e.g. jar://certs/CA_4_TESTcert2.crt>
DIGIDOC_CA_4_OCSP1_CN   =<enter common name for OCSP responder, e.g. TEST4
RESPONDER>
DIGIDOC_CA_4_OCSP1_CERT =<enter        cert        for        OCSP        responder,
e.g.jar://certs/TEST4Responder.crt>
DIGIDOC_CA_4_OCSP1_URL  =<enter URL for OCSP responder, e.g. http://www.testOCSP.ee>
```

Additional notes on Lithuanian ID tokens and CA usage with JDigiDoc

- However, since the Lithuanian ID tokens and certificates listed here do not form a part of the official DigiDoc framework, JDigiDoc support of them has been provided by SK separately (as developer of the JDigiDoc library).

- No cross-usability testing against the core DigiDoc library is being routinely carried out for the Lithuanian solutions, so compatibility with other DigiDoc components is not guaranteed.

- If OCSP confirmations are to be used against certificates issued by the above listed or other new CA's, 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, or provided in with TSL.

5.2. *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.

However, a legal and technical analysis of the practical usage of e-signatures shows that there are interoperability problems that currently limit the cross-border use of e-signatures.

The following is an overview of the features offered by JDigiDoc to support cross-border operability by using Trusted Service Provider Lists and implementing standard XadES profiles to produce qualified e-signatures (having a clear legal status under the e-signatures

Directive – i.e. the presumption of equivalence to a handwritten signature and the legal obligation of EU Member States to mutually recognize qualified certificates).

5.2.1. Trusted Service Provider Lists

In order to validate advanced e-signatures supported by qualified certificates, a receiving party first needs to check their trustworthiness. This means that the receiving party has to be able to verify whether the signature is

> 1. an advanced electronic signature
> 2. supported by a qualified certificate
> 3. issued by a supervised certification service provider.

The publicly available **Trusted Lists (TSL)** make it much easier for signature recipients to verify the e-signatures by complementing the data that can be retrieved from the e-signature and the qualified certificate and by providing also information on the supervised/ accredited status of Member States' certification service providers and their services.

JDigiDoc implements the TSLs for creation and validation of digital signatures. It can access a TSL directory specified in its configuration file and retrieve the information about the Certification Service Providers.

During the creation of a digital signature, the TSL data will be 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.

Note: the configuration file parameter **DIGIDOC_TSL_DIR** is used to specify the location of the TSL files for JDigiDoc; by default it will be set to *./jdigidoc/src/main/tsl.*

5.2.2. Supported BDOC profiles

As a new feature of the BDOC document format, JDigiDoc is offering various signature profiles which have been based on the XAdES profiles, differing in protection level offered.

Note: the current JDigiDoc development status for using the BDOC profiles is considered to be beta – meaning not every feature of every profile defined has been officially tested, supported. Also, using BDOC and its profiles with other DigiDoc libraries and core components is yet to be implemented.

The BDOC profiles correspond to the following XAdES forms (each profile including and extending the previous one:

BDOC profile	XAdES profile	Description
BES	XAdES (basic form)	No timestamps or OCSP responses **Note**: In the meaning of Estonian legislation this signature is not equivalent to handwritten signature.
T	XAdES-T (timestamp),	<SignatureValueTimeStamp>; Adding timestamp field to protect against

		repudiation
CL	XAdES-C (complete),	<CompleteCertificateRefs>, <CompleteRevocationRefs>; Adding references to verification data (certificates and revocation lists) to the signed documents to allow off-line verification and verification in future (but does not store the actual data);
TM	XAdES-X-L (extended long-term)	<CompleteCertificateRefs>, <CompleteRevocationRefs>, <CertificateValues> <RevocationValues> Adding actual certificates and revocation lists to the signed document to allow verification in future even if their original source is not available; uses time marking.
TS	XAdES-X-L (extended long-term),	<SignatureValueTimeStamp>, <CompleteCertificateRefs>, <CompleteRevocationRefs>, <CertificateValues> <RevocationValues> Adding actual certificates and revocation lists to the signed document to allow verification in future even if their original source is not available; uses time stamping.
TM-A **(functionality yet to be implemented)**	XAdES-A (archival),	<CompleteCertificateRefs>, <CompleteRevocationRefs>, <CertificateValues> <RevocationValues> <ArchieveTimeStamp> Adding possibility for periodical time marking (e.g. each year) of the archived document to prevent compromise caused by weakening signature during long-time storage period
TS-A **(functionality yet to be implemented)**	XAdES-A (archival),	<SignatureValueTimeStamp>, <CompleteCertificateRefs>, <CompleteRevocationRefs>, <SigAndRefsTimeStamp>, <CertificateValues> <RevocationValues>, <ArchieveTimeStamp> Adding possibility for periodical time stamping (e.g. each year) of the archived document to prevent compromise caused by weakening signature during long-time storage period

Additional notes on BDOC profile usage:

- The default profile used with BDOC format is TM.

- A different default profile can be set in the JDigiDoc configuration file, see Section 3.5. Configuring JDigiDoc, under Default BDOC profile.

- A different profile can also be specified during each signature's creation when using the JDigiDoc utility tool. The command line syntax to be used is:

```
For BDOC profile T:
 > java ee.sk.test.jdigidoc -ddoc-new BDOC T -ddoc-add c:\temp\test1.txt
text/plain –ddoc-sign 12345-ddoc-out c:\temp\test_T.bdoc

For BDOC profile CL:
> java ee.sk.test.jdigidoc -ddoc-new BDOC CL -ddoc-add c:\temp\test1.txt
text/plain –ddoc-sign 12345-ddoc-out c:\temp\test_CL.bdoc

For BDOC profile TM:
> java ee.sk.test.jdigidoc -ddoc-new BDOC TM -ddoc-add c:\temp\test1.txt
text/plain –ddoc-sign 12345-ddoc-out c:\temp\test_CL.bdoc

For BDOC profile TS:
> java ee.sk.test.jdigidoc -ddoc-new BDOC TS -ddoc-add c:\temp\test1.txt
text/plain –ddoc-sign 12345-ddoc-out c:\temp\test_CL.bdoc

For BDOC default profile (TM, unless set otherwise in config file):
> java ee.sk.test.jdigidoc -ddoc-new BDOC -ddoc-add c:\temp\test1.txt
text/plain –ddoc-sign 12345-ddoc-out c:\temp\test_default_TM.bdoc
```

5.3. Interoperability testing

5.3.1. 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 OpenXAdES/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
- SigAndRefsTimeStamp
- CertificateValues

- RevocationValues

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

5.3.2. DigiDoc framework cross-usability tests

Since JDigiDoc is a part of the OpenXAdES/DigiDoc framework, automated interoperability tests have been carried out between its libraries for C and Java.

The interoperability tests were executed through the **command line utility tools of both libraries**:

	For C library **(library/utility tool = abbreviation)**	**For Java library** **(library/utility program name= abbreviation)**
For .ddoc testing	libdigidoc/cdigidoc = **d**	JDigiDoc/ee.sk.test.jdigidoc= **j**
For .bdoc testing	libdigidocpp/digidoc-tool= **d**	JDigiDoc/ee.sk.test.jdigidoc= **j**
For .cdoc testing	libdigidoc/cdigidoc= **c**	JDigiDoc/ee.sk.test.jdigidoc= **j**

The different operating systems used in the cross-usability tests included:

- Linux (Ubuntu, OpenSuse, Fedora)
- Mac
- Windows

Test Suite 1

For example, in Test suite 1 for .ddoc, digitally signed documents were:

- created in the specified format (e.g. DIGIDOC-XML 1.3)
- created and signed using one library's command line tool (j for JDigiDoc or d for cdigidoc)
- verified using the other library's command line tool (d or j)
- all tests executed within one operating system.

Test suite 1 for .ddoc (DIGIDOC-XML) - lib j vs. lib d - within same OS - 1 smart card	Create_Add file_Sign	Verify_Extract
TC1	j	j
TC2	j	d
TC3	d	d
TC4	d	j

Sample command line options used:	```Create :```
	```-ddoc-new <version/profile>```
	```-ddoc-out <ddoc file>```
	
	```Add file:```
	```-ddoc-in <ddoc file>```
	```-ddoc-add <source data/input```
		```Verify:```
		```-ddoc-in <ddoc file>```
		```-ddoc-validate```
		
		```Extract:```
		```-ddoc-in <ddoc file>```
		```-ddoc-extract```

```
file> <text/plain>                    <extract_file_marker>
-ddoc-out <ddoc file>                 <tmp_data/output file>

Sign :
-ddoc-in <ddoc file>
-ddoc-sign <pin2> <test> <> <>
<> <> <> <correct_slot=0>
-ddoc-out <ddoc file>
```

Test Suite 2

In Test suite 2 for .ddoc, the digitally signed documents from previous Test suite 1 were:

- verified and signed again using one library's command line tool (j or d)

- verified again the other library's command line tool (d or j)

- tests were executed in a different operating system from Test suite 1 tests.

Test suite 2 for .ddoc (DIGIDOC-XML) - lib j vs. lib d - input from diff OS - 2 smart cards	Verify1	Add Signature	Verify2
TC1	d	j	d
TC2	j	d	j
Sample command line options used:	Verify: -ddoc-in <ddoc file> -ddoc-validate	Sign : -ddoc-in <ddoc file> -ddoc-sign <pin2> <test> <> <> <> <> <correct_slot=1> -ddoc-out <ddoc file>	Verify: -ddoc-in <ddoc file> -ddoc-validate

Test Suite 3

In Test suite 3 for .bdoc, digitally signed documents were:

- created in the specified format (e.g. BDOC TM)

- created and signed using one library's command line tool (j for JDigiDoc or d for digidoc-tool)

- verified using the other library's command line tool (d or j)

- tests were executed within one operating system

Test suite 3 for .bdoc (BDOC TM) - lib j vs. lib d - within same OS - 1 smart card	Create_Add file_Sign	Verify_Extract
TC1	j	j
TC2	j	d
TC3	d	d
TC4	d	j
Sample command line options used:	Create : -ddoc-new <version/profile> <signature_mechanism=TM> -ddoc-out <ddoc file> Add file: -ddoc-in <ddoc file> -ddoc-add <source data/input file>	Verify: -ddoc-in <ddoc file> -ddoc-validate Extract: If j, then using: -ddoc-in <ddoc file> -ddoc-extract

```
<text/plain>                        <extract_file_marker=D0>
-ddoc-out <ddoc file>               <tmp_data/output file>
                                    If d, then using:
Sign :                              open <ddoc file> --
If j, then using:                   extractAll=tmp_data
-ddoc-in <ddoc file>
-ddoc-sign <pin2> <test> <> <>
<> <> <correct_slot=0>
-ddoc-out <ddoc file>
If d, then using: create --
file=<source data/input file> --
<pin=pin2> <ddoc file>
```

Test Suite 4

In Test suite 4 for .bdoc, the digitally signed documents from previous Test suite 3 were:

- verified and signed again using one library's command line tool (j for JDigiDoc or d for digidoc-tool)

- verified again using the other library's command line tool (d or j)

- tests were executed in a different operating system from Test suite 3 tests.

Test suite 4 for .ddoc (BDOC TM) - lib j vs. lib d - input from diff OS - 2 smart cards	Verify1	Add Signature	Verify2
TC1	d	j	
TC2	j	d	
Sample command line options used:	Verify: -ddoc-in <ddoc file> -ddoc-validate	Sign : If j, then using: -ddoc-in <ddoc file> -ddoc-sign <pin2> <test> <> <> <> <> <> <correct_slot=1> -ddoc-out <ddoc file> If d, then using: create --file=<source data/input file> -- <pin=pin2> <ddoc file>	Verify: -ddoc-in <ddoc file> -ddoc-validate

Test Suite 5

In Test suite 5 for .cdoc, the digitally signed documents were:

- encrypted using one library's command line tool (j for JDigiDoc or c for cdigidoc)

- decrypted using the other library's command line tool (c or j)

- tests were executed within one operating system, using a single smart card for retrieving certificates needed for encrypting and decrypting.

Test suite 5 for .cdoc (encrypted digidoc) - lib j vs. lib c - within same OS - 1 smart card	Encrypt	Decrypt
TC1	j	j
TC2	j	c
TC3	c	c
TC4	c	j

Sample command line options used:	Encrypt:
	If j, then using:
	`-cdoc-recipient <pem file>`
	`-cdoc-encrypt-sk <input file>`
	
	If c, then using:
	`-encrecv <pem file>`
	`-encrypt-file <input file> <text/plain>`
		Decrypt, step 1 (output to .ddoc):
		If j, then using:
		`-cdoc-in`
		`<tmp_data/in_file_name_wo_ext.cdoc>`
		`-cdoc-decrypt-sk <pin1>`
		`<tmp_data/in_file_name_wo_ext.decrypted-tools_first_letter(tool).ddoc>`
		
		If d, then using:
		`-decrypt-file`
		`<tmp_data/#{in_file_name_wo_ext}.cdoc>`
		`<tmp_data/#{in_file_name_wo_ext}.decrypted-tools_first_letter(tool).ddoc> <pin1>`
		
		Decrypt, step 2 (extraction from .ddoc):
		`-ddoc-in`
		`<tmp_data/in_file_name_wo_ext}.decrypted-tools_first_letter(tool).ddoc>`
		`-ddoc-extract <extraxt_file_matker=D0>`
		`<tmp_data/in_file_name_wo_ext.decrypted-tools_first_letter(tool)>`

5.3.3. JDigiDoc API's usage 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 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-sign <pin-code>`	`SignatureProductionPlace(String city, String state, String country, String zip);`
	
	`SignatureFactory.getCertificate(int token, String pin);`
	
	`SignedDoc.prepareSignature(X509Certificate cert, String[] roles, SignatureProductionPlace adr);`
	
	`Signature.setProfile(String profile);`
	
	`Signature.calculateSignedInfoDigest();`
	
	`Signature.calculateSignedInfoXML();`
	
	`SignatureFactory.sign(byte[] digest, int token, String pin, Signature sig);`
	
	`Signature.setSignatureValue(byte[] sigval);`
	
	`Signature.getConfirmation();`

`-ddoc-in <input-digidoc-file>`	`DigiDocFactory.readSignedDoc(String inFile);`
`-ddoc-in-stream`	`SAXDigiDocFactory.readSignedDocFromStreamOfType(InputStream is, boolean isBdoc);`
`-ddoc-out <output-file>`	`SignedDoc.writeToFile(File outputFile);`
`-ddoc-out-stream`	`SignedDoc.writeToStream(OutputStream os);`

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 have been tested the most thoroughly.**JDigiDoc utility's command**	Called JDigiDoc API method(s)
-ddoc-list	SignedDoc.countSignatures(); DigiDocVerifyFactory.verifySignature(SignedDoc sdoc, Signature sig, ArrayList lerrs); Get methods of SignedDoc, Datafile, Signature, KeyInfoclasses.
-ddoc-validate	SignedDoc.validate(boolean bStrong); SignedDoc.countSignatures(); Get methods of SignedDoc, Signature, KeyInfo classes.
-ddoc-notarize	Signature.getConfirmation(); Signature.verify(SignedDoc sdoc, boolean checkDate, boolean demandConfirmation); SignedDoc.countSignatures(); 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);
-mid-sign <phone-no> <per-code> [[<country>(EE)] [<lang>(EST)] [<service>(Testing)] [<manifest>] [<city> <state> <zip>]]	DigiDocServiceFactory.ddsSign(SignedDoc sdoc, String sIdCode, String sPhoneNo, String sLang, String sServiceName, String sManifest, String sCity, String sState, String sZip, String sCountry, StringBuffer sbChallenge); DigiDocServiceFactory.ddsGetStatus(SignedDoc sdoc, String sSesscode);
-cdoc-in <input-encrypted-file>	EncryptedDataParser.readEncryptedData(String fileName);
-cdoc-list	Get methods of EncryptedData, EncryptedKey and EncryptionProperty classes.
-cdoc-validate	EncryptedData.validate();
-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>	SignedDoc(String format, String version); SignedDoc.addDataFile(File inputFile, String mime, String contentType);

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 have been tested the most thoroughly.**JDigiDoc utility's command**	Called JDigiDoc API method(s)

	`SignedDoc.readFile(File inFile);`
	
	`DataFile.setBase64Body(byte[] data);`
	
	`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>`	`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>`	`SignatureFactory.getAuthCertificate(int token, String pin);`
	
	`EncryptedData.decrypt(int nKey, int token, String pin);`

`-cdoc-decrypt-sk <pin> <output-file>`	`SignatureFactory.getAuthCertificate(int token, String pin);`
	
	`EncryptedData.decrypt(int nKey, int token, String pin);`
	
	`EncryptedData.getData();`
	
	`DigiDocFactory.readSignedDoc(String fileName);`
	
	`SignedDoc.getDataFile(int idx);`
	
	`DataFile.getBodyAsStream();`

`-cdoc-decrypt-stream <input-file> <recipient> <pin> <output-file>`	`EncryptedStreamParser.decryptStreamUsingRecipientName(InputStream dencStream, OutputStream outs, int token, String pin, String recipientName);`
`-cdoc-decrypt-pkcs12 <keystore-file> <keystore-passwd> <keystore-type> <output-file>`	`EncryptedData.decryptPkcs12(int nKey, String keystore, String storepass, String storetype);`

`-cdoc-decrypt-pkcs12-sk <keystore-file> <keystore-passwd> <keystore-type> <output-file>`	`EncryptedData.decryptPkcs12(int nKey, String keystore, String storepass, String storetype);`
	
	`EncryptedData.getData();`
	
	`DigiDocFactory.readSignedDoc(String fileName);`

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 have been tested the most thoroughly.**JDigiDoc utility's command**	**Called JDigiDoc API method(s)**

	`SignedDoc.getDataFile(int idx);`
	
	`DataFile.getBodyAsStream();`

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_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
#   CANONICALIZATION_FACTORY_IMPL = ee.sk.digidoc.factory.DOMCanonicalizationFactory
DIGIDOC_TSLFAC_IMPL        =         ee.sk.digidoc.tsl.DigiDocTrustServiceFactory
CRL_FACTORY_IMPL           =         ee.sk.digidoc.factory.CRLCheckerFactory
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
DIGIDOC_DF_CACHE_DIR    =         /tmp

# default digest type for new signatures - SHA-1, SHA-224, SHA-256, SHA-512
DIGIDOC_DIGEST_TYPE     =         SHA-256
# for .ddoc always SHA-1

# default digest type for all other digests - SHA-1, SHA-256 or SHA-512 - only for BDOC
DIGIDOC_DEFAULT_DIGEST   =        SHA-256
# for .ddoc always SHA-1

# digidoc default profile for BDOC format
DIGIDOC_DEFAULT_PROFILE =         TM
# TM = Qualified BDOC signature with time-marks
# available BDOC profiles are BES, T, CL, TM, TS

# PKCS#11 module settings - change this according to your signature device!!!
DIGIDOC_SIGN_PKCS11_DRIVER       =        /Library/OpenSC/lib/opensc-pkcs11.so
# for Estonian ID cards in Windows: DIGIDOC_SIGN_PKCS11_DRIVER = esteid-pkcs11
# for AID cards (GPK8000): DIGIDOC_SIGN_PKCS11_DRIVER = pk2privXAdES-XL.SCOK/SK/
DIGIDOC_SIGN_PKCS11_WRAPPER      =        PKCS11Wrapper
DIGIDOC_VERIFY_ALGORITHM =        RSA//


# VS: log4j config file - change this!!!
DIGIDOC_LOG4J_CONFIG    =         ./jdigidoc/src/main/resources/SignatureLogging.properties
DIGIDOC_TSL_DIR =         ./jdigidoc/src/main/tsl

# OCSP responder URL - change this!!!
DIGIDOC_OCSP_RESPONDER_URL       =        http://www.openxades.org/cgi-bin/ocsp.cgi
```

```
#   DIGIDOC_OCSP_RESPONDER_URL = http://ocsp.sk.ee
#DIGIDOC_OCSP_RESPONDER_URL = http://xades-portal.etsi.org/protected/capso/OCSP?ca=LevelBCAOK

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

# MI-ID signing
DDS_URL =          https://www.openxades.org:8443
#   DDS_URL = https://digidocservice.sk.ee
DDS_POLLFREQ     =        5
DDS_TRUSTSTORE   =        ./jdigidoc/src/main/resources/openxades.jks
DDS_STOREAPASS   =        openxades
DDS_STORETYPE    =        JKS

# Your HTTP proxy if necessary - change this!!!
#DIGIDOC_PROXY_HOST = <put-your-proxy-hostname-here>
#DIGIDOC_PROXY_PORT = <proxy-port>

# 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>

# Set this to true if you want jdigidoc to use ca certs registered in jdigidoc.cfg
# If you want jdigidoc to use only TSL-s set this to false.
DIGIDOC_USE_LOCAL_TSL     =        TRUE

# VS: 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      =        3
DIGIDOC_CA_1_NAME        =        AS Sertifitseerimiskeskus
DIGIDOC_CA_1_TRADENAME   =        SK
DIGIDOC_CA_1_CERTS       =        13
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/TEST-SK.crt
DIGIDOC_CA_1_CERT8       =        jar://certs/EECCRCA.crt
DIGIDOC_CA_1_CERT9       =        jar://certs/ESTEID-SK 2011.crt
DIGIDOC_CA_1_CERT10      =        jar://certs/EID-SK 2011.crt
DIGIDOC_CA_1_CERT11      =        jar://certs/TEST EECCRCA.crt
DIGIDOC_CA_1_CERT12      =        jar://certs/TEST ESTEID-SK 2011.crt
DIGIDOC_CA_1_CERT13      =        jar://certs/TEST EID-SK 2011.crt


DIGIDOC_CA_2_NAME        =        Nacionalinis sertifikavimo centras
DIGIDOC_CA_2_TRADENAME   =        NSC
DIGIDOC_CA_2_CERTS       =        7
DIGIDOC_CA_2_CERT1       =        jar://certs/lt-root.pem
DIGIDOC_CA_2_CERT2       =        jar://certs/lt-root-cr.pem
DIGIDOC_CA_2_CERT3       =        jar://certs/lt-issuing.pem
DIGIDOC_CA_2_CERT4       =        jar://certs/lt-policy.pem
```

```
DIGIDOC_CA_2_CERT5       =       jar://certs/NSC_Root_DF.cer
DIGIDOC_CA_2_CERT6       =       jar://certs/NSC_IssCA_DF.cer
DIGIDOC_CA_2_CERT7       =       jar://certs/NSC_PolCA_DF.cer
#DIGIDOC_CA_2_CERT1 = jar://certs/nacionalinis_ROOTCA.pem.cer
#DIGIDOC_CA_2_CERT2 = jar://certs/nacionalinis_POLICYCA.pem.cer
#DIGIDOC_CA_2_CERT3 = jar://certs/nacionalinis_ISSUINGCA.pem.cer


DIGIDOC_CA_3_NAME        =       VI Registru Centras RCSC
DIGIDOC_CA_3_TRADENAME   =       RCSC
DIGIDOC_CA_3_CERTS       =       3
DIGIDOC_CA_3_CERT1       =       jar://certs/vi-root.pem
DIGIDOC_CA_3_CERT2       =       jar://certs/vi-issuing.pem
DIGIDOC_CA_3_CERT3       =       jar://certs/vi-policy.pem

#DIGIDOC_CA_CERT8 = jar://certs/RootCAOK.pem
#DIGIDOC_CA_CERT8_URI = http://xades-portal.etsi.org/protected/capso/certs/RootCAOK.crt
#DIGIDOC_CA_CERT9 = jar://certs/LevelACAOK.pem
#DIGIDOC_CA_CERT9_URI = http://xades-portal.etsi.org/protected/capso/certs/LevelACAOK.crt
#DIGIDOC_CA_CERT10 = jar://certs/LevelBCAOK.pem
#DIGIDOC_CA_CERT10_URI = http://xades-portal.etsi.org/protected/capso/certs/LevelBCAOK.crt

# VS: 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       =       14

DIGIDOC_CA_1_OCSP1_CA_CN =       EID-SK TEST
DIGIDOC_CA_1_OCSP1_CA_CERT       =       jar://certs/EID-SK.crt
DIGIDOC_CA_1_OCSP1_CN    =       EID-SK OCSP RESPONDER
DIGIDOC_CA_1_OCSP1_CERT =        jar://certs/EID-SK OCSP 2006.crt
DIGIDOC_CA_1_OCSP1_URL   =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_CA_1_OCSP2_CA_CN =       EID-SK
DIGIDOC_CA_1_OCSP2_CA_CERT       =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP2_CN    =       EID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP2_CERT =        jar://certs/EID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP2_URL   =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP3_CA_CN =       ESTEID-SK TEST
DIGIDOC_CA_1_OCSP3_CA_CERT       =       jar://certs/ESTEID-SK.crt
DIGIDOC_CA_1_OCSP3_CN    =       ESTEID-SK OCSP RESPONDER
DIGIDOC_CA_1_OCSP3_CERT =        jar://certs/ESTEID-SK OCSP 2005.crt
DIGIDOC_CA_1_OCSP3_URL   =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_CA_1_OCSP4_CA_CN =       ESTEID-SK
DIGIDOC_CA_1_OCSP4_CA_CERT       =       jar://certs/ESTEID-SK 2007.crt
DIGIDOC_CA_1_OCSP4_CN    =       ESTEID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP4_CERT =        jar://certs/ESTEID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP4_URL   =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP5_CA_CN =       KLASS3-SK
DIGIDOC_CA_1_OCSP5_CA_CERT       =       jar://certs/KLASS3-SK.crt
DIGIDOC_CA_1_OCSP5_CN    =       KLASS3-SK OCSP RESPONDER
DIGIDOC_CA_1_OCSP5_CERT =        jar://certs/KLASS3-SK OCSP 2009.crt
DIGIDOC_CA_1_OCSP5_URL   =       http://ocsp.sk.ee

# Since 20.01.2007 issued ID-card certificates validity confirmation
# service
DIGIDOC_CA_1_OCSP6_CN    =       ESTEID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP6_CERT =        jar://certs/ESTEID-SK 2007 OCSP.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

# EID certificates (for example Mobile-ID certificates) issued since
# 20.01.2007 validity confirmation service
DIGIDOC_CA_1_OCSP7_CN    =       EID-SK OCSP RESPONDER 2007
DIGIDOC_CA_1_OCSP7_CERT  =       jar://certs/EID-SK 2007 OCSP.crt
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_CN    =       ESTEID-SK 2007 OCSP RESPONDER 2010
DIGIDOC_CA_1_OCSP8_CERT  =       jar://certs/ESTEID-SK 2007 OCSP 2010.crt
DIGIDOC_CA_1_OCSP8_CA_CERT  =       jar://certs/ESTEID-SK 2007.crt
DIGIDOC_CA_1_OCSP8_CA_CN =       ESTEID-SK 2007
DIGIDOC_CA_1_OCSP8_URL   =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP9_CERT  =       jar://certs/EID-SK 2007 OCSP 2010.crt
DIGIDOC_CA_1_OCSP9_CN    =       EID-SK 2007 OCSP RESPONDER 2010
DIGIDOC_CA_1_OCSP9_CA_CERT  =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP9_CA_CN =       EID-SK 2007
DIGIDOC_CA_1_OCSP9_URL   =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP10_CERT =       jar://certs/EID-SK 2007 OCSP.crt
DIGIDOC_CA_1_OCSP10_CN   =       EID-SK 2007 OCSP RESPONDER
DIGIDOC_CA_1_OCSP10_CA_CERT  =       jar://certs/EID-SK 2007.crt
DIGIDOC_CA_1_OCSP10_CA_CN    =       EID-SK 2007
DIGIDOC_CA_1_OCSP10_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP11_CERT =       jar://certs/ESTEID-SK OCSP 2005.crt
DIGIDOC_CA_1_OCSP11_CN   =       ESTEID-SK OCSP RESPONDER 2005
DIGIDOC_CA_1_OCSP11_CA_CERT  =       jar://certs/ESTEID-SK.crt
DIGIDOC_CA_1_OCSP11_CA_CN    =       ESTEID-SK
DIGIDOC_CA_1_OCSP11_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP12_CERT =       jar://certs/TEST-SK OCSP 2005.crt
DIGIDOC_CA_1_OCSP12_CN   =       TEST-SK OCSP RESPONDER 2005
DIGIDOC_CA_1_OCSP12_CA_CERT  =       jar://certs/TEST-SK.crt
DIGIDOC_CA_1_OCSP12_CA_CN    =       TEST-SK
DIGIDOC_CA_1_OCSP12_URL  =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_CA_1_OCSP13_CERT =       jar://certs/SK OCSP 2011.crt
DIGIDOC_CA_1_OCSP13_CN   =       SK OCSP RESPONDER 2011
DIGIDOC_CA_1_OCSP13_CA_CERT  =       jar://certs/EECCRCA.crt
DIGIDOC_CA_1_OCSP13_CA_CN    =       EE Certification Centre Root CA
DIGIDOC_CA_1_OCSP13_URL  =       http://ocsp.sk.ee

DIGIDOC_CA_1_OCSP14_CERT =       jar://certs/TEST SK OCSP 2011.crt
DIGIDOC_CA_1_OCSP14_CN   =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_CA_1_OCSP14_CA_CERT  =       jar://certs/TEST EECCRCA.crt
DIGIDOC_CA_1_OCSP14_CA_CN    =       TEST of EE Certification Centre Root CA
DIGIDOC_CA_1_OCSP14_URL  =       http://www.openxades.org/cgi-bin/ocsp.cgi

# OCSP or CRL selectors
DIGIDOC_CERT_VERIFIER    =       OCSP
DIGIDOC_SIGNATURE_VERIFIER      =       OCSP
DIGIDOC_TSA_URL =       http://tsp.iaik.tugraz.at/tsp/TspRequest
#   DIGIDOC_TSA_URL = http://xades-portal.etsi.org/protected/tsp/TspRequest
#   DIGIDOC_TSA_URL = http://tsp.iaik.at/tsp

# CRL settings
```

```
# VS not required if you don't use CRL-s
CRL_USE_LDAP    =       FALSE
CRL_FILE=       esteid.crl
CRL_URL =       http://www.sk.ee/crls/esteid/esteid.crl
CRL_SEARCH_BASE =       cn=ESTEID-SK,ou=ESTEID,o=AS Sertifitseerimiskeskus,c=EE
CRL_FILTER      =       (certificaterevocationlist;binary=*)
CLR_LDAP_DRIVER =       com.ibm.jndi.LDAPCtxFactory
CRL_LDAP_URL    =       ldap://194.126.99.76:389
CRL_LDAP_ATTR   =       certificaterevocationlist;binary
CRL_PROXY_HOST  =       cache.eypsise
CRL_PROXY_PORT  =       8080


# Encryption settings
# Compression mode of data before encryption. Possible values: 0 – always compress, 1 – never
compress, 2 – best effort
DENC_COMPRESS_MODE      =       0
# DENC_COMPRESS_MODE     =       1
# DENC_COMPRESS_MODE     =       2


# Software token usage settings
# Set these parameters when using software tokens
DIGIDOC_KEYSTORE_FILE   =       <your-PKCS#12-keystore-file>
DIGIDOC_KEYSTORE_TYPE   =       PKCS12
DIGIDOC_KEYSTORE_PASSWD =       <your-keystore-password>
# Additional parameters for digital signing with software tokens
# KEY_USAGE_CHECK        = FALSE
# DIGIDOC_SIGNATURE_SLOT = 0
```