CDigiDoc Programmer’s Guide

Document Version: 3.7

Library Version: 3.7

Last update: 22.01.2013

1. Document versions

Document information	
Created on	22.01.2013
Reference	CDigiDoc Programmer's Guide
Receiver	Sertifitseerimiskeskus AS
Author	Veiko Sinivee, Kersti Üts, Kristi Uukkivi
Version	3.7

Version information		
Date	**Version**	**Changes**
27.03.2006	2.2.5	The latest version of "DigiDoc C library" created by Veiko Sinivee
03.02.2012		Initial draft by KnowIT for the new version based on v2.2.5
22.02.2012	3.6	Updated to 3.6 version
22.05.2012	3.6.1	Revised configuration, certificates' usage and CDigiDoc utility program's description
22.01.2013	3.7	Updated to 3.7 version: updated instructions of PKCS#12 (software token) usage; removed EMBEDDED content type support, added description of signing and encryption/decryption operations in memory; added description of signature verification settings; added API description of decrypting large files, added description of using CNG API and minidriver for signature creation.

Table of contents

2. Introduction

This document describes CDigiDoc (also known as LibDigiDoc) – the C library for OpenXadES/DigiDoc system. It is a basic building tool for creating applications handling digital signatures and their verification.

COM library of DigiDoc system is described in a separate document, see [12].

The digitally signed files are created in "DigiDoc format" (with .ddoc file extension), compliant to XML Advanced Electronic Signatures (XAdES), technical standard published by European Telecommunication Standards Institute (ETSI). CDigiDoc is also capable of encrypting/decrypting files (signed or unsigned), according to W3C XML Encryption Recommendation (XML-ENC)

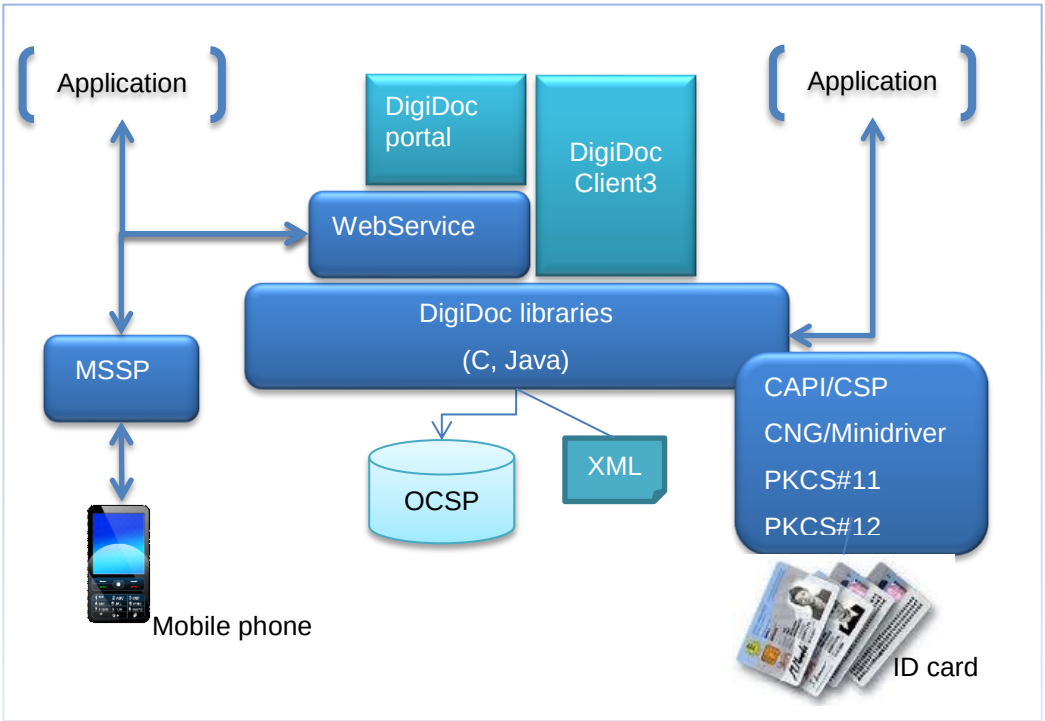
.

This document covers the following information about CDigiDoc:

- 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 CDigiDoc. 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 CDigiDoc, including sample use cases.
- Section 5 covers the currently supported tokens and CA's which have been tested with CDigiDoc. The section also gives an overview of interoperability testing results and lists the API functions that have been tested with the CDigiDoc utility program.
- Appendix 1 provides a sample CDigiDoc configuration file.

2.1. About DigiDoc

CDigiDoc 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.

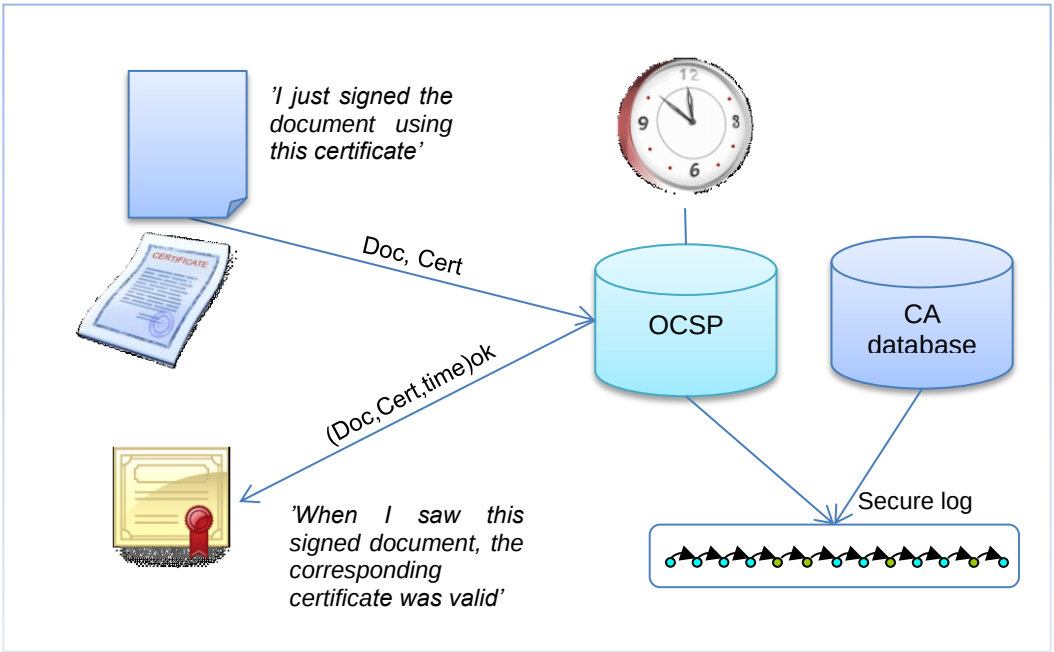
This proof is obtained in the format of Online Certificate Status Protocol (OCSP) response and stored within the signed document. Furthermore, (hash of the) created signature is sent within the OCSP request and received back within the response. This allows interpreting of the positive OCSP response 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. CDigiDoc is implementing a subset of these standards.

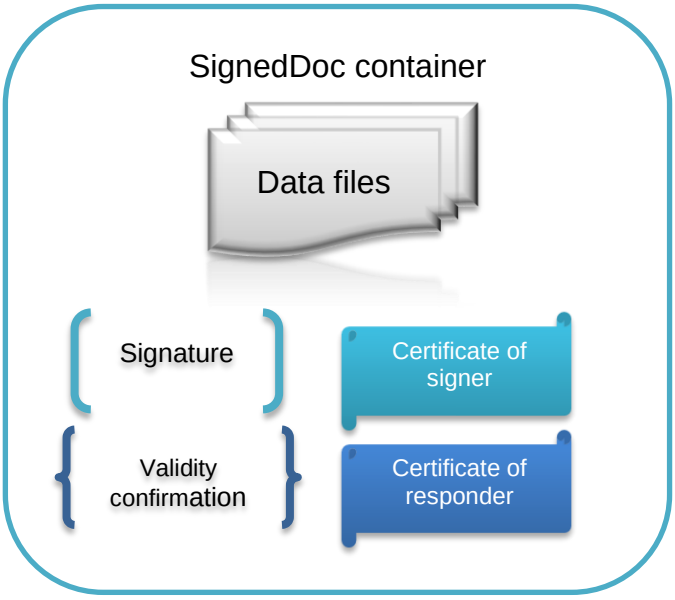
In order to comply with the security model described above, the XAdES profile **XAdES-X-L** is used in the DigiDoc system but "**time-marks**" are used instead of "time-stamps" – signing (and certificate validation) time comes with OCSP response.

This profile:

- allows for incorporating following signed properties
  - Certificate used for signing
  - Signing time
  - Signature production place
  - Signer role or resolution

- incorporates full certificate validity information within the signature
  - OCSP response
  - 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 DIGIDOC-XML document format to be used.

The DIGIDOC-XML document format (currently supported version 1.3) is fully conforming to XAdES standard (note however that not every single detail allowed in XAdES standard is supported).

DigiDoc system uses file extension **.ddoc** to distinguish digitally signed files according to the described file format. Syntax of the .ddoc file is described in a separate document in detail (see [6]).

The DIGIDOC-XML document's container is a single XML file which may contain embedded data file(s) and signature(s). It is possible to add data files to the container by:

- embedding binary data in base64 encoding (EMBEDDED_BASE64 mode),
- embedding pure text or XML – no longer supported (EMBEDDED mode),
- adding only reference to an external file – no longer supported (DETACHED mode).

SHA-1 digest type is supported and set automatically.

3. Overview

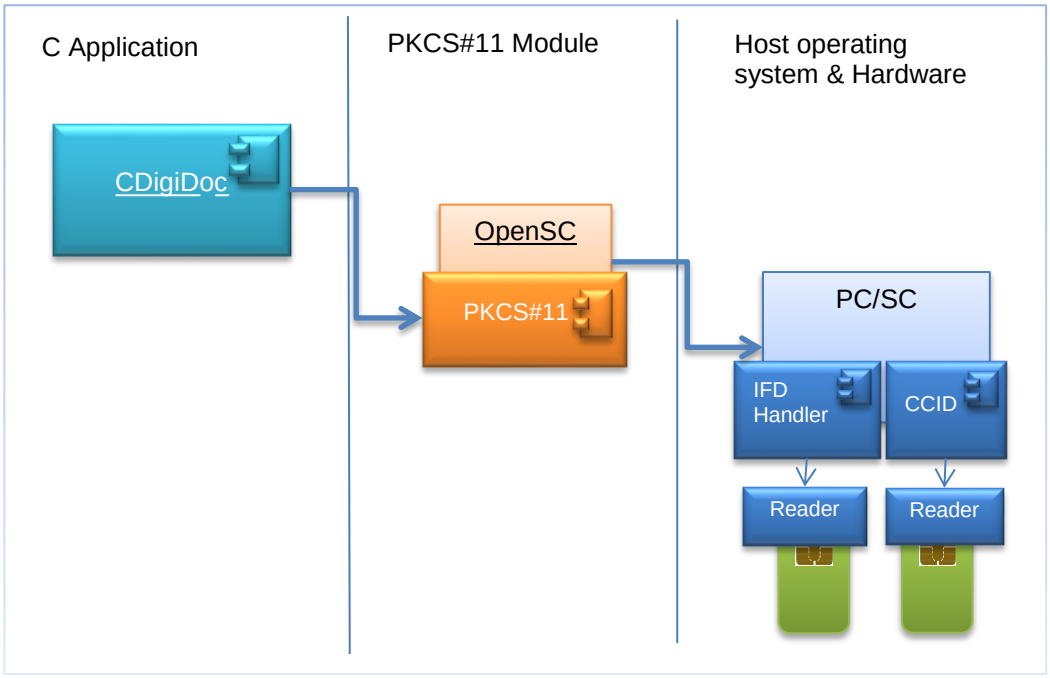
The following section describes the CDigiDoc library's architecture, configuring possibilities and examples of using it in C programs.

CDigiDoc is a library in C programming language offering the following functionality:

- Creating files in supported DigiDoc formats:
  - **DIGIDOC-XML 1.3**
- Digitally **signing** the DigiDoc files using smart cards or other supported cryptographic tokens.
- Adding **time marks** and **validity confirmations** to digital signatures using OCSP protocol.
- **Verifying** the digital signatures.
- Digital **encryption and decryption** of the DigiDoc files.

Note: older DigiDoc file formats SK-XML, DIGIDOC-XML 1.1 and DIGIDOC-XML 1.2 are supported only for backward compatibility in case of digital signature verification and data file extraction operations (creating new files and adding signatures is no longer supported).

The library supports using PKCS#11 and PKCS#12 cryptographic tokens. Example of using PKCS#11 module is given in the following figure.



4 Sample CDigiDoc 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

Note that in case of Windows environment, there can be two instances of the library installed concurrently. If you download and install the library's distribution package then it is stored to "c:\Program Files\Estonian ID Card Development\libdigidoc" directory by default. However, if you have DigiDoc Client3 program installed then the library is also included in its installation files ("c:\Program Files\Estonian ID Card" directory by default). Note that conflicts could occur between the two installations.

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://id.ee/public/DigiDoc_format_1.3.pdf)
[7] XML-ENC	http://www.w3.org/TR/xmlenc-core/
[8] DigiDocService Specification	EN: http://sk.ee/upload/files/DigiDocService_spec_eng.pdf ET: http://www.sk.ee/upload/files/DigiDocService_spec_est.pdf
[9] ESTEID	ESTEID Card Certification Policy (http://sk.ee/upload/files/SK-CP-ESTEID-3_2_en.pdf) Certificates on identity cards of Republic of Estonia (http://sk.ee/upload/files/ESTEID_profiil_en-3_3.pdf)
[10] COM library	COM library programmer's guide (http://id.ee/index.php?id=35784)
[11] Release notes	CDigiDoc library's release notes (http://id.ee/index.php?id=35782)

3.2. Terms and acronyms

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.
CAPI	Microsoft CryptoAPI, Cryptographic Application Programming Interface. API for implementing cryptographic operations in Windows operating systems
CNG	Cryptography API: Next Generation. Updated version of Microsoft CryptoAPI (CAPI).
CSP	Microsoft Crypto Service Provider. Software library that implements Microsoft CryptoAPI (CAPI)
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".
Minidriver	A device driver for controlling interaction with an identity token in Windows operating systems.
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.
PKCS#11	RSA Laboratories Cryptographic Token Interface Standard
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

CDigiDoc depends on the libraries listed below.

Base Component	Description
OpenSSL	Version 1.0.0 or newer. Source code is available from: http://www.openssl.org/
libxml2	Version 2.7.7 or newer. Source code is available from: http://www.xmlsoft.org/
Zlib	Version 1.2.4 or newer. Source code is available from: http://zlib.net/
iconv	Version 1.9.2 or newer. Source code is available from: http://www.gnu.org/software/libiconv/

3.4. Configuring CDigiDoc

CDigiDoc uses functions in DigiDocConfig.h/c source files for reading configuration data from property files. Sample configuration files are included in the library's installation package.

In Windows environment, the configuration file is named **digidoc.ini**, in Linux environment the file is named **digidoc.conf**.

Configuration settings may be loaded from different configuration files if the respective files are provided in system. Every subsequent configuration file complements the already present parameter values (i.e. doesn't delete the previous entries). CDigiDoc looks for configuration files in the following sequence:

1) system directory - C:\Windows\digidoc.ini in case of Windows environment or /etc/digidoc.conf in case of Linux. Specifies global settings that have effect on all the users of the computer.
2) user's home directory - c:\Users\<username>\digidoc.ini or /home/<username>/.digidoc.conf (notice the '.'), according to your environment.
3) current working directory - digidoc.ini or digidoc.conf, according to your environment.
4) in case of utility program, the configuration file that is provided with –config command.
5) in case of Windows environment, the Windows registry entries.

It is also possible to use a different configuration file location than the default. In that case, the configuration file's full filename and path should be passed to initConfigStore() function defined in DigiDocConfig.h or in case of CDigiDoc utility program, the file's location should be passed to the program with "–config" parameter (see section 4 for more information).

Note that if a configuration file is passed directly to initConfigStore() function or CDigiDoc utility program then this file is used over other files that might be stored in the default location(s).

For a sample configuration file provided with CDigiDoc, 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

PKCS#11 driver 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	Comments
DIGIDOC_DEFAULT_DRIVER	Specifies the default PKCS#11 driver library that is used to communicate with the smart card. ***1***
DIGIDOC_DRIVERS	Number of PKCS#11 drivers registered in the configuration file. Only one PKCS#11 driver at a time should be registered in a configuration file. ***1***
DIGIDOC_DRIVER_1_NAME	Name of the registered PKCS#11 driver library ***OpenSC***
DIGIDOC_DRIVER_1_DESC	PKCS#11 driver's description ***OpenSC projects PKCS#11 driver***
DIGIDOC_DRIVER_1_FILE	PKCS#11 driver library's filename ***opensc-pkcs11.dll (used in Windows environment)*** *# opensc-pkcs11.so (used in Linux environment)*

OCSP responder settings

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

The default OCSP responder has been set to http://ocsp.sk.ee which can be used with real-life Estonian ID cards.

Parameter	Description
DIGIDOC_OCSP_URL	OCSP responder address ***http://ocsp.sk.ee***

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 be 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 and password that have been issued to you for this purpose.

Parameter	Description
SIGN_OCSP	Specifies if OCSP requests are signed or not. Possible values: true – signed; false – not signed. ***false***
DIGIDOC_PKCS_FILE	Specifies your access certificate's PKCS#12 container location and filename, e.g.

	C:\temp\369787.p12d
DIGIDOC_PKCS_PASSWD	Specifies your access certificate's PKCS#12 container's password, e.g. ***m15eTGpA***

HTTP proxy settings*

*only necessary if using a proxy to access internet. Please note that configuring the following proxy settings has only been tested with DigiDoc Client3 program.

Parameter	Description
USE_PROXY	Specifies whether proxy is used. Possible values: true – used; false – not used. ***false***
DIGIDOC_PROXY_HOST	Specifies the proxy hostname, e.g. ***proxy.example.net***
DIGIDOC_PROXY_PORT	Specifies the proxy port, e.g. ***8080***
DIGIDOC_PROXY_USER	Specifies proxy server's username
DIGIDOC_PROXY_PASS	Specifies proxy server's password

CA certificates

The CA certificates are used to check the signer's certificate's validity.

By default, the Estonian CA's certificates (both live and test certificates) have been registered in the CDigiDoc configuration file. The live CA and OCSP certificate files have been included in the CDigiDoc distribution but the test certificate files haven't. In order to use the test certificates, you need to install them separately (the installation package is accessible from https://installer.id.ee/media/windows/Eesti_ID_kaart_testsertifikaadid.msi).

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

Parameter	Description
CA_CERT_PATH	Location of CA certificates. Supported Estonian CA certificates are included in CDigiDoc's installation package and will be located in the installation directory, e.g. ***C:\Program Files\Estonian ID Card Development\libdigidoc\certs***
CA_CERTS	Number of CA certificates registered in the configuration file, e.g. ***16***
CA_CERT_1 … CA_CERT_*n*	Name of a certificate file, e.g. ***ESTEID-SK 2007.crt***
CA_CERT_1_CN … CA_CERT_*n*_CN	Certificate's common name, e.g. ***ESTEID-SK 2007***

OCSP responder certificates

The following details should be provided for each OCSP Responder when OCSP responses are used in signature creation and verification.

The DIGIDOC_OCSP_RESPONDER_CERT_*n*_URL parameter is optional and has to be specified only in case of OCSP responder certificates which are used for testing purposes. In case of OCSP responders that correspond to test certificates registered in the CDigiDoc configuration file, the OpenXAdES OCSP Responder URL has been provided (http://www.openxades.org/cgi-bin/ocsp.cgi). For more information on using the OpenXAdES testing environment, please refer to http://www.openxades.org/tryitout.html.

Parameter	Description
DIGIDOC_OCSP_RESPONDER_CERTS	Number of OCSP Responder certificates registered in the configuration file, e.g. ***18***
DIGIDOC_OCSP_RESPONDER_CERT_1 … DIGIDOC_OCSP_RESPONDER_CERT_*n*	OCSP Responder certificate file's name, e.g. ***EID-SK OCSP 2006.crt***
DIGIDOC_OCSP_RESPONDER_CERT_1_1 … DIGIDOC_OCSP_RESPONDER_CERT_*n*_*n*	Additional certificate for the OCSP Responder, can be used if the alternative certificate is about to expire and new certificate is not yet valid, e.g. ***EID-SK OCSP.crt***
DIGIDOC_OCSP_RESPONDER_CERT_1_CN … DIGIDOC_OCSP_RESPONDER_CERT_*n*_CN	Name of the specific OCSP responder, e.g. ***EID-SK OCSP RESPONDER***
DIGIDOC_OCSP_RESPONDER_CERT_1_CA … DIGIDOC_OCSP_RESPONDER_CERT_*n*_CA	Name of the CA for the specific OCSP responder, e.g. ***EID-SK***
DIGIDOC_OCSP_RESPONDER_CERT_1_URL … DIGIDOC_OCSP_RESPONDER_CERT_*n*_URL	Address for the OCSP responder, has to be specified in case of OCSP responders for test certificates, e.g. ***http://www.openxades.org/cgi-bin/ocsp.cgi***

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 CDigiDoc utility program, "always compress" mode is used by default.

Debugging settings

Parameter	Description
DEBUG_LEVEL	Specifies the amount of debugging information printed out during execution. Possible value range: 0 – 9, e.g. ***3***
DEBUG_FILE	Full filename and path of debugging log file. If the parameter is set then debugging output is written to the specified file, e.g. ***c:\Temp\debug.log*** Note that the directory has to exist before debugging, otherwise the file is not created.

Signature verification settings

Parameter	Description
CHECK_OCSP_NONCE	Specifies if the OCSP response's nonce field's ASN.1 structure is checked during signature verification. By default, the value is set to false in order to support verification of DigiDoc files created with JDigiDoc library's version below v3.7. ***false*** # true
CHECK_SIGNATURE_VALUE_ASN1	Specifies if the signature value's ASN.1 structure is

	checked during signature verification. By default, the ASN.1 structure is checked. ***true*** # false

Data file content type setting

Parameter	Description
EMBEDDED_XML_SUPPORT	Specifies if JDigiDoc allows handling ddoc files that contain payload data as pure text or XML (data file content has been added in EMBEDDED mode). Should be used only to add backward compatibility for reading and validating EMBEDDED ddoc files. By default, EMBEDDED content mode is not supported (expected to produce a respective error message). Possible values are: false – not supported, true – supported.

Configuring software token usage

CDigiDoc supports using software tokens (PKCS#12 files) for creating technical signatures and decrypting files. Different configuration settings can be used for CDigiDoc library's versions v3.6 and v3.7.

1. Using the native PKCS#12 support (CDigiDoc v3.7 and above)

   No additional configuration settings have to be applied in case of decrypting with software token.

   In case of digital signing with software token, apply the following settings in CDigiDoc configuration file:

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

   Note that when verifying signatures that are created with the parameter value "KEY_USAGE_CHECK=0", an error message "Error: 39 - Signer's cert does not have non-repudiation bit set!" is produced.

2. Using an external sst module (CDigiDoc v3.6)

   The configuration described below sets up a software-based implementation of PKCS#11 and enables using PKCS#12 files instead of a physical smart card and reader.

   Your certificate and the accompanying private key have to be stored in separate files. You can extract the files from your PKCS#12 software token's container (.pfx file) by using OpenSSL.

Configuration for Windows environment (tested with Windows XP and Windows 7):

a. Download Scriptable Soft Token (SST) module from http://software.merit.edu/sst/, (sst.dll file).

b. Create directory c:\sst. Copy your PKCS#12 certificate and key files and sst.dll to this directory.

c. Create configuration file c:\sst.conf with the following content:

```
PIN = <pin to access your private key>

TOKEN_NAME = "<token name>"
TOKEN_MANUFACTURE_ID = "<manufacturer>"
TOKEN_MODEL = "<token model>"
TOKEN_SN = "<serial number>"
```

```
CERT = 1, <subject-name>, c:\sst\<your-cert>.crt, c:\sst\<your-key>.key

#EOF
```

Please note that:

- You can set token name, manufacturer and model values as you like.
- Serial number should be a positive integer.
- The file sst.conf has to be directly in c:\ root directory.

d. Set the following parameter value in CDigiDoc configuration file digidoc.ini:

```
DIGIDOC_DRIVER_1_FILE=c:\sst\sst.dll
```

Configuration for Linux environment (tested with Ubuntu):

a. Download soft-pkcs11 source code from http://people.su.se/~lha/soft-pkcs11/ and decompress the source.

b. Change your current directory to the top level of the source directory and run the following commands:

```
> ./configure
> make
> [switch to root user]
> cd soft-pkcs11
> make install
> [switch to current user]
```

As a result, soft-pkcs11.so file is created in /usr/lib directory.

c. Create rc file to home directory /home/<user>/.soft-token.rc with the following content:

```
<alias>\t<token name>\t<your-cert>\t<your-key>
```

Please note that:

- You can set alias and token name values as you like. Alias indicates the short version of the token's name. Note that space characters are allowed in token name but not in alias.
- "\t" should be replaced with an actual tabulator character.
- <your-cert> and <your-key> fields should contain the names of your certificate and private key files and absolute paths to the files in your file system.

For example, contents of the file could be:

```
test    test-cert    /home/tester/my_cert.cer    /home/tester/my_key.key
```

d. Set the following parameter value in CDigiDoc configuration file digidoc.conf:

```
DIGIDOC_DRIVER_1_FILE=/usr/lib/soft-pkcs11.so
```

Configuration for digital signing (for both Windows and Linux environments):

For creating digital signatures with software tokens, set the following additional parameter values in CDigiDoc configuration file:

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

Please note that when verifying signatures that are created with the parameter value "KEY_USAGE_CHECK=0", an error message "Error: 39 - Signer's cert does not have non-repudiation bit set!" is produced.

3.5. CDigiDoc architecture

The CDigiDoc library consists of three kinds of components:

- **Data structures** – declarations of data structures can be found in file DigiDocLib.h.
- **Constants** – a number of constants are used by the library, including error codes. Their definitions can be found in files DigiDocLib.h and DigiDocError.h.
- **Functions** – defined in *.c files of the library. Functions of public interest have been declared in file DigiDocLib.c.

For additional information about the functions and data structures of CDigiDoc library, see the full API description that is included in the CDigiDoc library's installation package, in directory /documentation/api.

3.6. Digital signing

CDigiDoc library offers creating, signing and verification of digitally signed documents, according to XAdES (ETSI TS101903) and XML-DSIG standards. In the next chapters a short introduction is given on the main API calls used to accomplish the above mentioned.

3.6.1. Initialization

Firstly, define the required structures:

```
SignedDoc* pSigDoc;
```

This structure reflects the file format of DigiDoc. All other relevant structures are part of this basic structure.

```
DataFile* pDataFile;
```

One DataFile structure corresponds to one original data file (file-to-be-signed) in DigiDoc container. One DigiDoc container can incorporate multiple data files. The data files are embedded in the DigiDoc container.

Initialize the library with the following function:

```
initDigiDocLib();
```

This ensures all OpenSSL library parameters are properly initialized.

3.6.2. Creating a DigiDoc document

DigiDoc structure should first be created in memory:

```
SignedDoc_new(&pSigDoc,
      DIGIDOC_XML_1_1_NAME, // format of the DigiDoc document
      DIGIDOC_XML_1_3_VER); // default version number
```

Values of the constants above are defined as "DIGIDOC-XML" and "1.3" (in DigiDocLib.h source file).

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

In the following sections, we add a data file and a signature to the DigiDoc structure before writing it into an output file.

3.6.3. Adding data files

Data files can be added to a DigiDoc container in two alternative ways:

- by reading the data from a file on the disk and using temporary files to store any intermediary data;
- by using only internal memory buffers (e.g. data to be added has been generated dynamically or has been read from a database).

1. Adding a data file by reading the file from disk and storing intermediary data in temporary files

   Firstly, call the function Datafile_new(). The function creates a new DataFile element and saves the original data file in DigiDoc container:

```
DataFile_new(&pDataFile, // data file to be added
      pSigDoc, // DigiDoc structure to which the data file is added
      NULL, // data file's id
      infile, // data file name and path
    CONTENT_EMBEDDED_BASE64, // file embedding option
      mime, // mime type of the data file
      0, NULL, 0, NULL, // optional parameters
      CHARSET_UTF_8); // fixed constant for DigiDoc character encoding
```

   Third parameter in the abovementioned function is a unique identification of the data file in the DigiDoc document. If value NULL is used then the library generates it automatically.

   Fourth parameter is the name of the data file. It is recommended to include full path in this parameter; the path is removed when writing the file to DigiDoc container.

   Fifth parameter reflects how data files are embedded in the DigiDoc container. The supported embedding option is `CONTENT_EMBEDDED_BASE64` (defined in DigiDocLib.h) – contents of the data file are encoded using base64-encoding before merging it into DigiDoc container.

   Sixth parameter is a MIME type of the data file. For example "application/msword" or "application/pdf", depending on the type of the data file.

   In most cases, the next four parameters should be left to the library to determine. The parameters determine:

   - size of the original file in bytes,
   - hash of the original file,
   - size of the hash of the original file,
   - type of hash algorithm (only SHA-1 is supported).

   To calculate the values of these four parameters, call the following function:

```
calculateDataFileSizeAndDigest(pSigDoc, pDataFile->szId, infile,
DIGEST_SHA1);
```

   This function reads the data file from disk, calculates and adds these four values to section pDataFile->szId based on file name given in the third parameter. `DIGEST_SHA1` is the only supported hash algorithm.

   It is possible to add additional extra XML attributes to the data file, function addDataFileAttribute() is used for that. For example:

```
addDataFileAttribute(pDataFile, "ISBN", "000012345235623465");
addDataFileAttribute(pDataFile, "Owner", "CEO");
```

The first parameter is a pointer to original file structure, followed by the attribute's name and value. The data is going to be added in UTF-8 encoding.

2. Adding data by using internal memory buffers (no data is stored on the disk)

   Use the DigiDocMemBuf structure to hold the data to be added. Note that the fields of DigiDocMemBuf structure should be initialised with value 0:

```
DigiDocMemBuf mbuf; // memory buffer to hold the data
mbuf.pMem = 0; // functions will assign allocated memory address here
mbuf.nLen = 0; // length of data in number of bytes
```

   It is possible to assign data to DigiDocMemBuf as follows:

   - ddocMemAssignData() – assigns data to DigiDocMemBuf memory buffer and releases previous data content if necessary; ddocMemAppendData() – appends data to memory buffer and increases its length accordingly.

     The functions are defined in source file DigiDocMem.h and can be used, for example, to assign data that has been read from database or generated dynamically.

   - function ddocReadFile() defined in DigiDocConfig.h – reads data from an input file and assigns it to DigiDocMemBuf memory buffer.

   Add the data in memory buffer to DigiDoc container with the following function (defined in DigiDocObj.h):

```
createDataFileInMemory(&pDataFile, // structure of the data file to be added
    *ppSigDoc, // DigiDoc structure to which the data file is added
    NULL, // data file's id
    infile, // data file name and path
    CONTENT_EMBEDDED_BASE64, // file embedding option
    mime, // mime type of the data file
    mbuf.pMem, // memory buffer's data
    mbuf.nLen); // memory buffer's size
```

   The function creates a new DataFile element, adds the data and its hash value (SHA-1 is supported) to DigiDoc container.

   The first six parameters of the function above have the same meaning as in function DataFile_new() described in the previous point.

   The last two input parameters specify the data in memory buffer and the buffer's size.

   After the new DataFile element has been created and added to the DigiDoc container, you can add additional attributes to it with function addDataFileAttribute() (described in the previous point).

   Memory of the data buffer should be freed after the data has been added:

```
ddocMemBuf_free(&mbuf);
```

3.6.4. Adding signatures

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.

Note: the functionality of adding signatures is no longer supported in case of older DigiDoc file formats SK-XML, DIGIDOC-XML 1.1 and DIGIDOC-XML 1.2.

SignatureInfo structure is needed to incorporate the necessary information about the signature before it can be created:

```
SignatureInfo* pSigInfo;
```

Signing can be done by using the function:

```
signDocumentWithSlotAndSigner(SignedDoc* pSigDoc, // SignedDoc structure
                                                  // to which the signature is added
      SignatureInfo** ppSigInfo, // SignatureInfo structure
      const char* pin, // pin2 in case of Estonian ID cards
      const char* manifest, // signer's role (optional)
      const char* city, // signature production place (optional)
      const char* state, const char* zip, const char* country,
      int nSlot, // specifies the signer's private key's slot
      int nOcsp, // add OCSP confirmation or not
      int nSigner, // signing module: 1->PKCS11, 2->CNG, 3->PKCS12
      const char* szPkcs12FileName); // PKCS#12 file name
```

Optional parameter **manifest** may be used to define the signer's role. Optional parameters **city**, **state**, **zip** and **country** may be used to specify the signature creation location.

Parameter **nSlot** indicates the sequence number (counting from zero) of the current signing certificate among all signature certificates on the identity token. In case of Estonian ID cards, there is one signature certificate and the signature slot value is 0 (which is also the default slot value). When operating with multiple smartcards on the same system then you may need to specify a different slot. By default, in this case, the signature slots are numbered as follows:

slot 0 – signature slot of the 1st smartcard

slot 1 – signature slot of the 2nd smartcard

In case of signing with a software token (PKCS#12 file) then firstly the appropriate configuration settings should be applied (see section 3.4, subsection "Configuring software token usage") and slot value 0 should be used.

Parameter **nOcsp** can be used to specify whether an OCSP confirmation is added to the signature or not. The default value is 1, meaning that OCSP confirmation is automatically added to the signature after its creation. Value 0 indicates that the confirmation is not added.

Parameter **nSigner** specifies the module that is used for accessing the signature token. Possible values for the parameter are as follows:

- int nSigner = 1 - signing is done via PKCS#11 module which is the default module for singing with smart card.
- int nSigner = 2 – Microsoft CNG API and minidriver for signing with smart card in Windows environment. A dialog window is opened for the user to choose the signing certificate and enter PIN code.
- int nSigner = 3 - signing is done via PKCS#12 module, to be used in case of creating signatures with software tokens (PKCS#12 files).

Parameter **szPkcs12FileName** should be specified only when signing with a software token via PKCS#12 module (i.e. the nSigner parameter has been set to 3).

The function signDocumentWithSlotAndSigner() creates a new SignatureInfo structure and adds information about the data files to be signed and optional metadata of the signature (role of the signer and signature production place) to the structure. Then the signature value is calculated and stored. Finally, an OCSP confirmation is added to the signature if the value of nOcsp parameter indicates it.

3.6.5. Adding an OCSP confirmation

OCSP protocol is used to get validity confirmation from OCSP Responder to prove that certificate was valid at the time of signing.

It is possible to add an OCSP confirmation to a signature during its creation with function signDocumentWithSlotAndSigner() - value of parameter nOcsp has to be set to "1" (as described in the previous section).

Alternatively, you can add the confirmation by calling out the appropriate function yourself:

```
// Get the SignatureInfo element of the signature to be confirmed
// according to the signature's identificator
pSignInfo = getSignatureWithId(pSigDoc, szSignId);
// Get the OCSP confirmation
notarizeSignature(pSigDoc, pSignInfo);
```

Information about the signer's certificate CA and its respective OCSP responder is retrieved from CDigiDoc's configuration file. CA and OCSP Responder data have to be registered in the configuration file and the respective certificates have to be stored in the file system.

Note: there are also alternative functions for adding an OCSP confirmation to a signature but it is recommended to use the notarizeSignature() function described above. The function enables handling cases when the signer's certificate and OCSP responder's certificate are issued by different CAs and if there are several CA certificates with matching CN names.

Note: when verifying a signature that has no OCSP confirmation, an error message "Error: 128 - Signature has no OCSP confirmation!" is produced.

3.6.6. Reading and writing DigiDoc documents

Reading DigiDoc documents

It is possible to read an existing DigiDoc document from a file stored in the file system or from an internal memory buffer (e.g. buffer that stores DigiDoc document's data fetched from a database).

1. Opening and reading a DigiDoc document from disk

   Use the function:

   ```
   int ddocSaxReadSignedDocFromFile(SignedDoc** ppSigDoc,// DigiDoc structure
         const char* szFileName, // input file
         int checkFileDigest,
         long lMaxDFLen);
   ```

   Parameter checkFileDigest is a flag indicating whether checking hash value(s) of original file(s) is required at the time of opening. Parameter lMaxDFLen can be used to specify the maximum size of DataFile content to be cached in memory.

2. Opening and reading DigiDoc document from a memory buffer

   Use the DigiDocMemBuf structure to hold the initial data (see also section "3.6.3 Adding data files", under the second point for additional information about initialising and using DigiDocMemBuf):

   ```
   DigiDocMemBuf mbuf; // memory buffer to hold the data
   ```

   Define a SignedDoc structure for holding the DigiDoc document's data that is read from buffer:

   ```
   SignedDoc* pSigDoc;
   ```

   Read the DigiDoc document from DigiDocMemBuf buffer by using the following function:

```
ddocSaxReadSignedDocFromMemory(&pSigDoc, // structure representing the
                                         //DigiDoc document
      mbuf.pMem, // memory buffer's data
      mbuf.nLen, // memory buffer's size
      mbuf.nLen + 1);//max size of DataFile content to be cached in memory
```

Writing DigiDoc documents

DigiDoc documents can be created in two alternative ways:

- creating the output (new or modified) DigiDoc document and writing it to a file on disk;
- creating the output (new or modified) DigiDoc document and storing it in internal memory buffer (no data is written to disk).

1. Writing the output DigiDoc document to a file on disk

   For creating a file in DigiDoc format and writing it to a file, the following function should be used:

   ```
   createSignedDoc(pSigDoc, // structure representing the DigiDoc document
         oldfile, // specifies existing DigiDoc file, if necessary
         outfile); // output file's name
   ```

   The "oldfile" parameter value can be set to NULL if you are creating a new DigiDoc document from scratch. If you have read in an existing DigiDoc document to modify it (e.g. add signature(s) or data file(s)) and now try to write it to an output file then you have to specify the existing DigiDoc file's path and filename in the "oldfile" parameter. Otherwise the data file contents from the existing DigiDoc file might not be copied to the new container.

2. Writing the output DigiDoc document to an internal memory buffer

   Use the DigiDocMemBuf structure for storing the output DigiDoc document's data (see also section "3.6.3 Adding data files", under the second point for additional information on initialising and using DigiDocMemBuf):

   ```
   DigiDocMemBuf* pMBuf; // output buffer
   ```

   Write the created or modified DigiDoc container to a memory buffer by using the following function:

   ```
   createSignedDocInMemory(SignedDoc* pSigDoc, // structure representing
                                               // the DigiDoc document
       const char* oldfile, // specifies existing DigiDoc file, if necessary
       DigiDocMemBuf* pMBuf); // memory buffer for storing the output data
   ```

   The "oldfile" parameter in the abovementioned function should be used according to the analogous parameter in createSignedDoc() function (described in the previous point).

Memory should be released after end of working with DigiDoc structure:

```
SignedDoc_free(pSigDoc);
```

This also releases memory that is used for keeping the data files.

After finishing work with CDigiDoc, then the last task is to shut down the library:

```
finalizeDigiDocLib();
```

3.6.7. Verify signatures and OCSP confirmations

You can verify a signature and its OCSP confirmation with the function:

```
int verifySignatureAndNotary(SignedDoc* pSigDoc,
             SignatureInfo* pSigInfo, const char* szFileName);
```

For example, after having read a DigiDoc document, verify its signatures as follows:

```
SignedDoc* pSigDoc;
SignatureInfo* pSigInfo;

initDigiDocLib();

// Open and read in the DigiDoc document from a file
err = ddocSaxReadSignedDocFromFile(&pSigDoc, inFile, 0, 0);

// Get the count of signatures
numberOfSignatures = getCountOfSignatures(pSigDoc);
for(counter = 0; counter < numberOfSignatures; counter++) {
      pSigInfo = getSignature(pSigDoc, counter);

// Verify the signature

      err = verifySignatureAndNotary(pSigDoc, pSigInfo, inFile);
}
SignedDoc_free(pSigDoc);
finalizeDigiDocLib();
```

3.7. Encryption and decryption

In addition to digital signing, CDigiDoc 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.

CDigiDoc 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 (PEM encoding is 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 CDigiDoc library, the certificates that can be used for encryption must have the value "Key Encipherment" included in "Key Usage" attribute field.

CDigiDoc 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.

However, it is possible to incorporate a number of data files in one encrypted document if they are firstly all added to a DigiDoc container and then encryption is performed for that container as for a single data object.

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

3.7.1. Composing encrypted documents

Note: for compatibility with other DigiDoc software components, it is recommended to place the data file to be encrypted inside a DigiDoc container before encrypting it. This way it is also possible to incorporate multiple data files into one encrypted document (i.e. if there is more than one data file in the DigiDoc container that is encrypted).

In order to compose an encrypted document you have to:

- create the DencEncryptedData structure first
- add all recipient info and other meta-information
- add the unencrypted data
- encrypt it, possibly compressing the data
- store it in a file or another medium.

The encryption method described is most suitable for small or medium sized data objects – all operations are done in memory.

Note that in order for the encrypted document to be compatible with other DigiDoc software components then the data file to be encrypted should be placed in a DigiDoc container before encryption (if the file is not originally a DigiDoc document).

Start composing a new encrypted document by defining the required data structures:

```
DEncEncryptedData** ppEncData;
```

The DEncEncryptedData structure refers to the <EncryptedData> element of an encrypted file and is the main structure that is used to store information which is needed for performing the encryption. Other structures that are used should be defined as follows:

```
DEncEncryptedKey* pEncKey; // transport key data for every recipient
DEncEncryptionProperty* pEncProperty; // property structure for storing
      // various metadata
```

Now create the DencEncryptedData structure with the following function:

```
dencEncryptedData_new(ppEncData,
      DENC_XMLNS_XMLENC, // fixed constant for XML namespace uri
      DENC_ENC_METHOD_AES128, // fixed constant for encryption method
            // algorithm uri
      0, 0, 0); // optional attributes, not needed with the current
            // encrypted document format
```

3.7.2. Adding recipient info and metadata

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 appropriate for data encryption must be used. In case of Estonian ID cards it is the authentication certificate.

Start adding recipient data by reading in the recipient's certificate (the certificate has to be in PEM format):

```
ReadCertificate(&pCert, certfile);
```

Function ReadCertificate() (defined in source file DigiDocCert.h) reads the certificate from a file in file system. Alternatively, you can also use functions ddocDecodeX509Data() (data is in binary format) and ddocDecodeX509PEMData() (data is in PEM (base64) format) to decode certificate data that is already in memory.

Encrypt the transport key with the receiver's certificate and store encrypted key in memory:

```
dencEncryptedKey_new(*ppEncData, &pEncKey,
      pCert, // reveiver's certificate
      DENC_ENC_METHOD_RSA1_5, // fixed constant for encryption method
      id, recipient, keyname, carriedkeyname); // optional attributes
```

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

You can add metadata about the CDigiDoc library that is used for creating the encrypted document and encrypted document's format and version:

```
dencMetaInfo_SetLibVersion(*ppEncData);
dencMetaInfo_SetFormatVersion(*ppEncData);
```

The name of the data file that is encrypted should be added to the DencEncryptedData structure by creating a new property:

```
dencEncryptionProperty_new(*ppEncData, &pEncProperty,
      0, 0, // property id and target. Can be omitted
      ENCPROP_FILENAME, // fixed constant, represents the data file's name
      getSimpleFileName(dataFile)); // data file's name should be added
             // without path
```

Note that the data file's name used in the previous example has to be in UTF-8 encoding. If necessary, you can convert it with function:

```
ddocConvertInput(const char* src, char** dest);
```

If the original file is a .ddoc file then you should specify its mime type and add the value to DencEncryptedData structure as a new property:

```
dencEncryptionProperty_new(*ppEncData, &pEncProperty, 0, 0,
      ENCPROP_ORIG_MIME, // name of the property: original mime type
      DENC_ENCDATA_TYPE_DDOC); // value of the property: ddoc document's
             // mime type
```

In case of DigiDoc document, mime type has to be specified as shown above so that it would be possible to decrypt the file later.

Constants that represent mime types have been defined in DigiDocEnc.h source file. In case of a DigiDoc document, use the constant

- `DENC_ENCDATA_TYPE_DDOC`

  which contains the value:

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

The value is assigned to property "MimeType" of the cdoc document. CDigiDoc library uses the property "MimeType" also to store the fact that the data has been packed with ZLIB algorithm before encryption. If data compression is used then the library assigns the value

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

to "MimeType" attribute which has also been defined as a constant:

- `DENC_ENCDATA_MIME_ZLIB`

CDigiDoc assigns this value when packing the data and if the "MimeType" attribute was not empty before then the previous value is stored in `<EncryptionProperty Name="OriginalMimeType">` sub element instead. If CDigiDoc reads a document with "MimeType" value defined by `DENC_ENCDATA_MIME_ZLIB` then at first it decompresses the decrypted data and then restores the original mime type if one is found.

If the original data file to be encrypted is a ddoc document then after adding the mime type property, you also need to "register" its contents:

```
dencOrigContent_registerDigiDoc(*ppEncData,
      pSigDoc); // SignedDoc structure representing the ddoc document
```

The function creates a new EncryptionProperty structure for every data file contained in the DigiDoc document and stores its name, size, mime type and id values for later use.

Note that you need to have the DigiDoc document kept in memory as a SignedDoc structure before using the function in the previous example. If you are encrypting an existing DigiDoc document (not creating it directly before encryption) then read the document in as described in section "3.6.6 Reading and writing DigiDoc documents".

3.7.3. Encryption and data storage

Before encrypting, you also need to add the actual data to be encrypted to DEncEncryptedData structure. Use the method:

```
dencEncryptedData_AppendData(DEncEncryptedData* pEncData,
      const char* data, // unencrypted data
      int len); // length of the data
```

Finally, encrypt the data with the following function:

```
dencEncryptedData_encryptData(DEncEncryptedData* pEncData,
      int nCompressOption); // compression option used before encryption
```

In the function above, three different constants can be used to specify compression option for the data to be encrypted:

- `DENC_COMPRESS_ALLWAYS` - data is compressed before encryption.
- `DENC_COMPRESS_BEST_EFFORT` - data will be compressed and if it results in reduced data size then the compressed data is encrypted. Otherwise it will be discarded and original data is encrypted with no compression.
- `DENC_COMPRESS_NEVER` - compression is not applied.

You can write the encrypted document to an output file with the function:

```
dencGenEncryptedData_writeToFile(DEncEncryptedData* pEncData, const char*
szFileName);
```

Note that it isn't necessary to use files to store encrypted data. It can be written to any output stream and used as required. In order to write the encrypted data to a memory buffer, do as follows:

```
DigiDocMemBuf mbuf; // output buffer
mbuf.pMem = 0; // functions will assign allocated memory address here
```

```
mbuf.nLen = 0; // length of data in number of bytes
dencGenEncryptedData_toXML(pEncData, // encrypted data structure
    &mbuf); // output buffer for storing the encrypted data
```

3.7.4. Parsing and decrypting

Firstly, define structure for holding the encrypted document's data that is going to be parsed:

```
DEncEncryptedData* pEncData;
```

There are two alternative options for decrypting documents, depending on the encrypted document's size.

1. Parsing and decrypting small encrypted documents.

   Encrypted document can be read in and parsed in two ways: by reading the encrypted file from disk or by reading the encrypted file's contents from an internal memory buffer (e.g. a buffer that holds the encrypted document's data fetched from a database).

   - Reading and parsing the encrypted document from file system:

     ```
     char* inFile; // input encrypted file
     dencSaxReadEncryptedData(&pEncData, // structure for holding the encrypted
                                         // document's data
         fileName); // input encrypted file's name
     ```

   - Reading and parsing encrypted document from a memory buffer

     Use the DigiDocMemBuf structure for holding the encrypted document's initial data (see also section "3.6.3 Adding data files", under the second point for additional information about initialising and using DigiDocMemBuf):

     ```
     DigiDocMemBuf mbuf; // data buffer structure
     ```

     Parse the encrypted document from the memory buffer with the following function:

     ```
     dencSaxReadEncryptedDataFromMemory(&pEncData, // structure for holding the
                                        // encrypted document's data
         &mbuf); // memory buffer with the initial data
     ```

   After parsing the document, data can be decrypted or displayed on screen. Decryption is a separate operation and is not automatically done during parsing.

   For decrypting, you need to find the correct EncryptedKey structure for the current recipient who is decrypting the data. If you use PKCS#11 identity token from a smart card for decryption, then do:

   ```
   dencEncryptedData_findEncryptedKeyByPKCS11(*ppEncData, &pEncKey);
   ```

   Now, data can be decrypted as follows:

   ```
   dencEncryptedData_decrypt(*ppEncData,
       pEncKey, // transport key
       pin); // pin1 code in case of Estonian ID cards
   ```

   The abovementioned functions are defined in source file DigiDocEnc.h. Function dencEncryptedData_decrypt() firstly decrypts the transport key with the recipient's pin code and then decrypts the data with the transport key. Data is decompressed, if necessary.

2. Parsing and decrypting large encrypted documents.

   In order to parse and decrypt large files, use the dencSaxReadDecryptFile() function (defined in DigiDocEncSAXParser.h):

   ```
   dencSaxReadDecryptFile(const char* szInputFileName, // encrypted file's name
       const char* szOutputFileName, // output (decrypted) file
   ```

```
        const char* szPin, // pin1 code in case of Estonian ID cards
        const char* szPkcs12File); // set to NULL in case of Estonian ID cards
```

The abovementioned function reads encrypted data from the specified input file, decrypts and possibly decompresses the data during parsing and writes the decrypted data to output file. Data is not kept in memory during decryption.

Parameter szPkcs12File indicates the PKCS#12 software token's file name and path, if decryption is done with a software token. The value should be set to NULL when using PKCS#11 driver (e.g. in case of Estonian ID cards).

Note: when decrypting files then it should be taken into account that for compatibility with other DigiDoc software components, it is recommended that the data file to be encrypted is placed inside a DigiDoc container before encryption. In this case, it is also necessary to extract the original data file(s) from DigiDoc container after decryption.

4. CDigiDoc utility

CDigiDoc library includes a command line utility program – cdigidoc.exe – which can be used to read, digitally sign, encrypt and decrypt files in OpenXadES format. Source code of the program is in cdigidoc.c file.

The general format is:

```
> cdigidoc [command(s)]
```

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

```
> cdigidoc -help
```

Output from all of the CDigiDoc utility program's commands is ended with the following information:

```
CDigiDoc|[error code or '0' in case of success]|[elapsed time in seconds]
```

Note that the error codes' definitions can be found in the file DigiDocError.h.

4.1. General commands

- **-? or –help** – displays help about command syntax.
- **-config <configuration-file>** - specifies the CDigiDoc 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 configuration file is looked up from default locations (see section "3.4 Configuring CDigiDoc" for more information).

Checking the certificate

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

Used for checking the chosen certificate's validity; returns an OCSP response from the certificate's CA's OCSP responder. Note that the command is currently not being tested.

If the certificate is valid, then the return code's (RC) value is 0. For example:

Verifying cert: MÄNNIK,MARI-LIIS,47101010033 --> RC :0

```
Sample: setting the configuration file when creating a new DigiDoc container
> cdigidoc –config c:\temp\digidoc.ini -new -add c:\temp\test1.txt
text/plain -out c:\temp\test1.ddoc

  Input:
```

```
- c:\temp\digidoc.ini – 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
```

4.2. Digital signature commands

- **-in <input-digidoc-file> -** reads in a DigiDoc file
- **-in-mem <input-digidoc-file> -** reads in a DigiDoc file. The operation is conducted "in memory", meaning that the data is read into a memory buffer and no intermediary data is written to temporary files on the disk.
- **-new –** creates a new DigiDoc container
- **-add <input-file> <mime-type> –** adds a data file to a DigiDoc container
- **-add-mem <input-file> <mime-type> -** adds a data file to a DigiDoc container. The operation is conducted "in memory", meaning that the data is read into a memory buffer and no intermediary data is written to temporary files on the disk.
- **-sign <pin-code> –** signs a DigiDoc file
- **-out <output-file> –** creates a DigiDoc file at the specified location
- **-out-mem <output-file> -** creates a DigiDoc file at the specified location. The operation is conducted "in memory", meaning that the data is read from and written to a memory buffer, no intermediary data is written to temporary files on the disk.
- **-list –** displays a DigiDoc file's content info and verifies signature(s)
- **-verify –** displays and verifies DigiDoc file's signature(s)
- **-extract <data-file-id> <output-file> –** extracts DigiDoc file's content
- **-get-confirmation <signature-id> –** adds an OCSP confirmation to a DigiDoc file's signature.
- **-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

-new [format] [version]

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

By using the optional parameter - version - with this command, you can specify an alternative **version** to be created.

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

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

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 data 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 reflects how the original files are embedded in the container: EMBEDDED_BASE64 (embedding binary data in base64 format) is supported and used by default.

Charset - UTF-8 encoding is supported and used by default.

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

Alternative version of the -add command. The operation is conducted "in memory", meaning that the data is read into a memory buffer and no intermediary data is written to temporary files on the disk.

-sign <pin-code> [[[manifest] [city] [state] [zip] [country]] [slot(0)] [ocsp(1)] [PKCS11/CNG/PKCS12] [pkcs12-file-name]]

Adds a digital signature to the digidoc document. Note that adding signatures to DigiDoc files in older formats SK-XML, DIGIDOC-XML 1.1 and DIGIDOC-XML 1.2 is no longer supported. You can use the command with the following parameters:

pin-code	PIN code of the identity token. In case of Estonian ID cards, PIN2 code is used for digital signing. Required when signing via PKCS#11 (the default module) and PKCS#12 module, optional in case of CNG API and minidriver (see also parameter "PKCS11/CNG/PKCS12" of the current command).
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
country	Country of origin. ISO 3166-type 2-character country codes are used (e.g. EE)
slot	Identifier of the signer's certificate's and private key's sequence number (counting from zero) among all signature certificates on an identity token. When operating for example with a single Estonian ID card (which contains one signature key) then the key can be found in slot 0 – which is used by default. The library makes some assumptions about PKCS#11 drivers and card layouts: - you have signature and/or authentication keys on the card - both key and certificate are in one slot - if you have many keys like 1 signature and 1 authentication key then they are in different slots - you can sign with signature key that has a corresponding certificate with "NonRepudiation" bit set. You may need to specify a different slot to be used when for example operating with multiple smart cards on the same system. In this case, the signature slots are counted as follows: - slot 0 – signature key of the 1st smartcard - slot 1 – signature key of the 2nd smartcard If the slot needs to be specified during signing, then the 5 previous optional parameters (manifest, city, state, zip, country) should be

	filled first (either with the appropriate data or as "" for no value).
ocsp	Specifies whether an OCSP confirmation is added to the signature that is being created. Possible values are 0 – confirmation is not added; 1 – confirmation is added. By default, the value is set to 1. Parameter value 0 can be used when creating a technical signature. Technical signature is a signature with no OCSP confirmation and no timestamp value.
PKCS11/CNG/PKCS12	Optional parameter to specify module that is used for accessing the signature token. Possible values are: - "PKCS11" - default module for signing with smart card - "CNG" - alternative module for signing with smart card, uses Microsoft CNG API and smart card's minidriver. A dialog window is opened for the user to choose a signing certificate and insert PIN code (i.e. the "pin-code" parameter of the current command may be left unspecified by inserting an empty string ""). - "PKCS12" - module for signing with a software token (PKCS#12 file). When signing with a software token then firstly, the appropriate configuration settings should be applied (see section 3.4, subsection "Configuring software token usage"). The current command's parameter "pin-code" must be set according to the PKCS#12 container's PIN code, parameter "ocsp" must be set to 0 and parameter's "pkcs12-file-name" value should be specified.
pkcs12-file-name	Used only when signing with software token (PKCS#12 file) via PKCS#12 module (i.e. if the previous parameter's value has been set to "PKCS12"). Specifies the software token's file name.

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

Invokes mobile signing of a ddoc 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 the 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

-out <output-file>

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

-out-mem <output-file>

Alternative version of the -out command. The operation is conducted "in memory", meaning that the data is read from and written to a memory buffer, no intermediary data is written to temporary files on the disk.

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)

```
> cdigidoc -new -add c:\temp\test1.txt text/plain -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

```
> cdigidoc -new -add c:\temp\test1.txt text/plain -sign 12345 -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
  - 12345               - id-card pin2
  - c:\temp\test1.ddoc  - container to be created
```

Sample: signing an existing DigiDoc container (adding signatures)

```
> cdigidoc -in c:\temp\test1.ddoc -sign 12345 -out c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc         - container to be signed
  - 12345                      - id-card pin2
  - c:\temp\test1.ddoc         - output (modified) digidoc container
```

Sample: using Mobile-ID for signing

```
> cdigidoc -new -add c:\temp\test1.txt text/plain –mid-sign +3706234566
41110170240 -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
  - +3706234566         - signer's mobile number
  - 41110170240         - signer's personal code
  - c:\temp\test1.ddoc  - container to be created
```

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

```
> cdigidoc -in c:\temp\test1.ddoc -add C:\temp\test3.txt text/plain -add
C:\temp\test4.txt text/plain -out c:\temp\test1.ddoc

Input:
```

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

Sample: signing an existing digidoc container via CAPI/CNG module
> cdigidoc -in c:\temp\test1.ddoc -sign "" "" "" "" "" "" 0 1 CNG -out
c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc  - unsigned container to be read and modified
  - ""                  - empty strings for PIN code and other optional
                parameter values (manifest, city, state, zip, country)
  - 0                   - signature slot
  - 1                   - OCSP confirmation is added
  - CNG                 - identifier of CAPI/CNG module usage
  - c:\temp\test1.ddoc  - output (modified) digidoc container
```

Sample commands for signing in memory

```
Sample: creating new DigiDoc file with signing, operation in memory

> cdigidoc -new -add-mem c:\temp\test1.txt text/plain -sign 12345 -out-mem
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
  - 12345               - id-card pin2
  - c:\temp\test1.ddoc  - container to be created

Sample: signing an existing DigiDoc container (adding signatures), operation
in memory
> cdigidoc -in-mem c:\temp\test1.ddoc -sign 12345 -out-mem
c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc          - container to be signed
  - 12345                       - id-card pin2
  - c:\temp\test1.ddoc          - output (modified) digidoc container
```

Sample commands for signing with technical signature

Technical signature is a signature with no OCSP confirmation or a signature created with a software token. Note that when verifying a signature that has no OCSP confirmation, an error message "Signature has no OCSP confirmation!" is produced. When verifying signature that is created with a software token, an error message "Signer's cert does not have non-repudiation bit set!" is produced.

```
Sample: signing an existing digidoc container with a technical signature
(via default (PKCS#11) module)
> cdigidoc -in c:\temp\test1.ddoc -sign 67890 "" "" "" "" "" 0 0 -out
c:\temp\test1.ddoc

Input:
```

```
  - c:\temp\test1.ddoc  - unsigned container to be read and modified
  - 67890               - PIN code
  - “”                  - empty strings for optional parameter values
  (manifest, country, state, city, zip)
  - 0                   - signature slot
  - 0                   - OCSP confirmation is not added
  - c:\temp\test1.ddoc  - output (modified) digidoc container

Sample: signing an existing digidoc container with a technical signature by
using a PKCS#12 software token (via PKCS#12 module)
> cdigidoc -in c:\temp\test1.ddoc -sign 67890 “” “” “” “” “” 0 0 PKCS12
c:\test\pkcs12.pfx -out c:\temp\test1.ddoc

Input:
  - c:\temp\test1.ddoc  - unsigned container to be read and modified
  - 67890               - software token's PIN code
  - “”                  - empty strings for optional parameter values
  (manifest, country, state, city, zip)
  - 0                   - signature slot
  - 0                   - OCSP confirmation identifier
  - PKCS12              - identifier of PKCS12 module
  - c:\test\pkcs12.pfx  - your software token's PKCS#12 container file
  - c:\temp\test1.ddoc  - output (modified) digidoc container
```

Reading DigiDoc files and verifying signatures

-in <input-digidoc-file>

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

-in-mem <input-digidoc-file>

Alternative version of the -in command. The operation is conducted “in memory”, meaning that the data is kept in memory buffers and no intermediary data is written to temporary files on the disk.

-list

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

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

  SignedDoc | <format-identifier> | <version>

  For example: SignedDoc | DIGIDOC-XML | 1.3

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

  DataFile | <file identifier> | <file name> | <file size in bytes> | <mime type> | <data file embedding option>

  For example: DataFile | D0 | test1.txt | 44 | text/plain | EMBEDDED_BASE64

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

  Signature | <signature identifier> | <signer's key info: last name, first name, personal code> | <verification return code> | <verification result>

  For example: Signature | S0 | MÄNNIK,MARI-LIIS,47101010033 | 0 | No errors

- **Signer's certificate information**

- **OCSP responder certificate information**

-verify

Returns signature **verification results** (if signatures exist):

- Signature | <signature identifier> | <signer's key info: last name, first name, personal code> | <verification return code ('0' for success)> | <verification result>

  For example: Signature | S0 | MÄNNIK,MARI-LIIS,47101010033 | 0 | No errors

Returns signer's certificate and OCSP Responder certificate information.

-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 (e.g. D0, D1…).

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
> cdigidoc -in c:\temp\test1.ddoc -list

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

Returns:
SignedDoc|DIGIDOC-XML|1.3
DataFile|D0|test1.txt|44|text/plain|EMBEDDED_BASE64
DataFile|D1|test2.txt|84|text/plain|EMBEDDED_BASE64

Sample: listing DigiDoc file's contents, signed
> cdigidoc -in c:\Temp\test1_s.ddoc -list

Input:
  - c:\temp\test1_s.ddoc – the digidoc file which contents are to be listed

Returns:
SignedDoc|DIGIDOC-XML|1.3
DataFile|D0|test1.txt|44|text/plain|EMBEDDED_BASE64
DataFile|D1|test2.txt|84|text/plain|EMBEDDED_BASE64
Signature|S0|MÄNNIK,MARI-LIIS,47101010033|0|No errors
/prints out signer's and OCSP responder's certificate data/

Sample: verifying DigiDoc file's signatures
> cdigidoc -in c:\Temp\test2.ddoc -verify

Input:
- C:\temp\test2.ddoc    – the digidoc file to be verified

Returns:
Signature|S0|MÄNNIK,MARI-LIIS,47101010033|0|No errors
/prints out signer's and OCSP responder's certificate data/

Sample: Extracting a data file from an existing DigiDoc file
> cdigidoc -in c:\temp\test1.ddoc -extract D0 c:\temp\test_ext.txt

Input:
- c:\temp\test1.ddoc    – the digidoc file to be extracted from
```

```
- D0                    - the data file ID to be extracted
- c:\temp\test_ext.txt - file for storing the extracted data
```

4.3. Encryption commands

- **-in <input-encrypted-file>** - reads in the specified encrypted input document
- **-in-mem <input-encrypted-file> -** reads in an encrypted file. The operation is conducted "in memory", meaning that the data is read into a memory buffer and no intermediary data is written to temporary files on the disk.
- **-out <output-decrypted-file> -** specifies the decrypted output document's name
- **-out-mem <output-decrypted-file> -** creates a decrypted output document at the specified location. The operation is conducted "in memory", meaning that the data is read from and written to a memory buffer, no intermediary data is written to temporary files on the disk.
- **-denc-list <input-encrypted-file>** - displays the encrypted document data and recipient's info.
- **-encrecv <certificate-file> -** adds recipient to an encrypted document
- **-encrypt-sk <input-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.
    - *-encrypt <input-file>* - used for encrypting small files, not recommended for compatibility with other DigiDoc software components.
    - *-encrypt-file <input-file> <output-file>* **-** used for encrypting large files, not recommended for compatibility with other DigiDoc software components.
- **-decrypt-sk <output-file> <pin> -** 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:
    - *-decrypt <output-file> <pin>* - used for decrypting small files in any original format.
    - *-decrypt-file <input-file> <output-file> <pin>* - used for decrypting large files in any original format.
    - *decrypt-hex <input-file> <key> <output-file> - used for testing decryption operation, Previously decrypted transport key value has to be provided.*

Reading encrypted files

-in <input-encrypted-file>

Input encrypted file (required) specifies the encrypted file's name.

-in-mem <input-encrypted-file>

Alternative version of the -in command. The operation is conducted "in memory", meaning that the data is read into a memory buffer and no intermediary data is written to temporary files on the disk.

-denc-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

```
> cdigidoc -denc-list c:\Temp\test1b.cdoc

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

Returns:
  EncryptedData|||http://www.isi.edu/in-
  noes/iana/assignments/mediatypes/application/zip|http://www.w3.org/2001/04
  /xmlenc#aes128-cbc
  LIBRARY|CDigiDoc|2.7.1.59
  FORMAT|ENCDOC-XML|1.0
  EncryptedKey||MÄNNIK,MARI-
  LIIS,47101010033|||http://www.w3.org/2001/04/xmlenc#rsa-1_5|OK
  EncryptionProperties|
  EncryptionProperty|||LibraryVersion|CDigiDoc|2.7.1.59
  EncryptionProperty|||DocumentFormat|ENCDOC-XML|1.0
  EncryptionProperty|||Filename|test1.txt
  EncryptionProperty|||OriginalMimeType|http://www.sk.ee/DigiDoc/v1.3.0/digi
  doc.xsd
  EncryptionProperty|||orig_file|c:\temp\test1.txt|44|application/file|D0
  EncryptionProperty|||OriginalSize|360
  EncryptionProperty|||OriginalMimeType|http://www.sk.ee/DigiDoc/v1.3.0/digi
  doc.xsd
```

Encrypting files

-encrecv <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 (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. 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. For example if CN = MÄNNIK,MARI-LIIS,41110212444, then recipient = MÄNNIK,MARI-LIIS,41110212444
KeyName	Sub-element <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	Sub-element <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.

-out <output-encrypted-file>

Output encrypted 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**.

-out-mem <output-encrypted-file>

Alternative version of the -out command. The operation is conducted "in memory", meaning that the data is read from and written to a memory buffer, no intermediary data is written to temporary files on the disk.

-encrypt-sk <input-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.

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

-encrypt <input-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.**

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

-encrypt-file <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 in 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**.

Command line samples for encrypting documents:

```
Sample: encrypting small doc (DigiDoc compatible, original in any format)
> cdigidoc -encrypt-sk c:\temp\test_Small.txt –out c:\Temp\test1.cdoc -
encrecv c:\temp\Rcert.cer MÄNNIK,MARI-LIIS,47101010033
```

```
Input:
- c:\temp\test_Small.txt – the input file to be encrypted
- c:\temp\test1.cdoc    - the encrypted file to be created
- c:\temp\Rcert.cer     – the recipient's certificate file
- MÄNNIK,MARI-LIIS,47101010033 - the recipient's ID (= certificate's CN)

Sample: encrypting small doc (not DigiDoc compatible, unless original doc
already in DIGIDOC-XML format)
> cdigidoc -encrypt c:\temp\test_Small.ddoc -out c:\Temp\test1.cdoc -encrecv
c:\temp\Rcert.cer

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)
> cdigidoc -encrypt-file c:\temp\test_Large.ddoc c:\Temp\test1.cdoc -encrecv
c:\temp\Rcert.cer

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: encrypting small doc for multiple recipients
> cdigidoc -encrypt-sk c:\temp\test1.txt –out c:\Temp\test1.cdoc -encrecv
c:\temp\R1cert.cer -encrecv c:\temp\R2cert.cer

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

Sample: encrypting small doc (DigiDoc compatible, original in any format),
operation in memory
> cdigidoc -encrypt-sk c:\temp\test_Small.txt –out-mem c:\Temp\test1.cdoc -
encrecv c:\temp\Rcert.cer MÄNNIK,MARI-LIIS,47101010033

Input:
- c:\temp\test_Small.txt – the input file to be encrypted
- c:\temp\test1.cdoc    - the encrypted file to be created
- c:\temp\Rcert.cer     – the recipient's certificate file
- MÄNNIK,MARI-LIIS,47101010033 - the recipient's ID (= certificate's CN)
```

Decrypting files

-decrypt-sk <input-file> <pin> [pkcs12-file] [slot(0)]

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

Input file (required) specifies the input file's name.

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

pkcs12-file (optional) specifies the PKCS#12 file if decrypting is done with a software token.

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

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

-decrypt <input-file> <pin> [pkcs12-file] [slot(0)]

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

Input file (required) specifies the input file's name.

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

pkcs12-file (optional) specifies the PKCS#12 file if decrypting is done with a software token.

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

-decrypt-file <input-file> <output-file> <pin> [pkcs12-file]

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

Input file (required) specifies the encrypted file to be decrypted.

Output file (required) specifies the output file name.

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

pkcs12-file (optional) specifies the PKCS#12 file if decrypting is done with a software token.

-decrypt-hex <input-file> <key> <output-file>

For testing purposes. Decryption of the input file can be done by providing transport key value that has previously been decrypted with the recipient's private key.

Input file (required) specifies the encrypted file to be decrypted.

Key (required) specifies transport key's value that has previously been decrypted with recipient's private authentication key. The key should be provided in hexadecimal format.

Output file (required) specifies the output file name.

Command line samples for decrypting documents:

```
Sample: decrypting small encrypted file, inside a DigiDoc container
> cdigidoc -decrypt-sk c:\Temp\test1_small.cdoc 1234 -out
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
> cdigidoc -decrypt c:\Temp\test1_small.cdoc 1234 -out 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
 > cdigidoc -decrypt-file c:\Temp\test1_large.cdoc c:\Temp\test1_d.ddoc 1234

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
 > cdigidoc -decrypt-sk c:\Temp\test1_small.cdoc 123456 c:\test\pkcs12.pfx -
out c:\Temp\test1_d.txt

Input:
  - c:\Temp\test1_small.cdoc    – the encrypted file to be decrypted
  - 123456                      - pin code of the software token
  - c:\test\pkcs12.pfx          - software token (PKCS#12 container) file
  - c:\temp\test1_d.txt         - the decrypted file to be created

Sample: decrypting, specifying slot value
> cdigidoc -decrypt-sk c:\Temp\test1_small.cdoc 1234 "" 1 -out
c:\Temp\test1_d.ddoc

Input:
  - c:\Temp\test1_small.cdoc    – the encrypted file to be decrypted
  - 1234                        – the PIN code of the recipient
  - ""                          - empty string for optional PKCS#12 file
parameter
  - 1                           - slot (sequence number) of the recipient's
decryption certificate on identity token
  - C:\temp\test1_d.ddoc        - the decrypted file to be created

Sample: decrypting small encrypted file, inside a DigiDoc container,
operation in memory
> cdigidoc -decrypt-sk c:\Temp\test1_small.cdoc 1234 -out-mem
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
```

4.4. Commands in CGI mode

CDigiDoc utility program can be used as a CGI program to add digital signature creation functionality to web sites.

Note: the CGI mode commands are not currently included in testing.

- **-calc-sign <cert-file> [<manifest>] [<city> <state> <zip> <country>]** – calculate hash of a digital signature. The certificate file has to be in PEM format, in a separate file. The calculated hash is displayed in console in base64 format.
- **-add-sign-value <sign-value-file> <sign-id> -** add a RSA-SHA1 signature. The signature has to be in base64 format in a separate file.
- **-del-sign <sign-id>** - remove a digital signature.
- **-cgimode [<output-separator>] -** output in CGI mode. Data sets in output are separated with the specified output separator symbol. '|' is used by default.
- **-consolemode -** output in console (not CGI) mode
- **-SAX -** use SAX parser
- **-XRDR -** use XmlReader parser

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

Token	Type	Description	Supported CDigiDoc 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 -	All CDigiDoc functionalities (authentication, signing, verification, encryption/decryption)

		EECCRCA)	
Digi-ID (since 2010)	Certificate –based PKI smart card	Estonian Digital ID card for use only in electronic environments	All CDigiDoc 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:
  - having a contract with SK and accessing the service from a specific IP address(es) – as practiced **by companies/services**
  - by having certificate for accessing the service and sending signed requests - as used **by private persons** for giving digital signatures; registering for the service is required and service is limited to 10 signatures per month
- CRL-s (mainly for backward compatibility)
- LDAP directory service (containing all valid certificates)

Supported SK live hierarchy chains

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

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

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

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

Supported SK test certificate hierarchy chains

Note: the following test certificates have been registered in the CDigiDoc configuration file but have not been included in the CDigiDoc distribution. In order to use the test certificates with CDigiDoc, you need to install them separately (the installation package is accessible from https://installer.id.ee/media/windows/Eesti_ID_kaart_testsertifikaadid.msi).

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

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

All of the above listed SK certificates can be downloaded from http://www.sk.ee/en/repository/certs/.

5.2. Interoperability testing

5.2.1. DigiDoc framework cross-usability tests

Since CDigiDoc 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 .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 file> <text/plain>`
	`-ddoc-out <ddoc file>`
	
	`Sign :`
	`-ddoc-in <ddoc file>`
	`-ddoc-sign <pin2> <test> <> <> <> <> <> <correct_slot=0>`
	`-ddoc-out <ddoc file>`
		`Verify:`
		`-ddoc-in <ddoc file>`
		`-ddoc-validate`
		
		`Extract:`
		`-ddoc-in <ddoc file>`
		`-ddoc-extract <extract_file_marker> <tmp_data/output 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 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)>

5.2.2. CDigiDoc API's usage in CDigiDoc utility program

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

CDigiDoc utility's command	**Called CDigiDoc API method(s)**
-check-cert	**ReadCertificate**(X509 **x509, const char *szCertfile); **ddocVerifyCertByOCSP**(X509* pCert, OCSP_RESPONSE **ppResp); **ddocCertGetSubjectCN**(X509* pCert, DigiDocMemBuf* pMemBuf);
-in <input-ddoc-file>	**ConfigItem_lookup_int**(const char* key, int defValue); **ddocSaxReadSignedDocFromFile**(SignedDoc** ppSigDoc, const char* szFileName, int checkFileDigest, long lMaxDFLen);
-in <input-encrypted-file>	**ConfigItem_lookup_int**(const char* key, int defValue); **dencSaxReadEncryptedData**(DEncEncryptedData** ppEncData, const char* szFileName);

CDigiDoc utility's command	Called CDigiDoc API method(s)
-new	**ConfigItem_lookup**(const char* key); **SignedDoc_new**(SignedDoc **pSignedDoc, const char* format, const char* version);
-add <input-file> <mime-type>	**ddocConvertInput**(const char* src, char** dest); **getFullFileName**(const char* szFileName, char* szDest, int len); **DataFile_new**(DataFile **newDataFile, SignedDoc* pSigDoc, const char* id, const char* filename, const char* contentType, const char* mime, long size, const byte* digest, int digLen, const char* digType, const char* szCharset); **calculateDataFileSizeAndDigest**(SignedDoc* pSigDoc, const char* id, const char* filename, int digType);
-sign <pin-code>	**signDocumentWithSlotAndSigner**(SignedDoc* pSigDoc, SignatureInfo** ppSigInfo, const char* pin, const char* manifest, const char* city, const char* state, const char* zip, const char* country, int nSlot, int nOcsp, int nSigner, const char* szPkcs12FileName);
-out <output-ddoc-file>	**createSignedDoc**(SignedDoc* pSigDoc, const char* szOldFile, const char* szOutputFile);
-out <output-encrypted-file>	**dencGenEncryptedData_writeToFile**(DEncEncryptedData* pEncData, const char* szFileName);
-list (in case of ddoc file)	**getCountOfDataFiles**(const SignedDoc* pSigDoc); **getDataFile**(const SignedDoc* pSigDoc, int nIdx); Functions of –verify command.
-list (in case of encrypted file)	**dencMetaInfo_GetLibVersion**(DEncEncryptedData* pEncData, char* szLibrary, int nLibLen, char* szVersion, int nVerLen); **dencMetaInfo_GetFormatVersion**(DEncEncryptedData* pEncData, char* szFormat, int nFormat, char* szVersion, int nVersion);
-verify	**getCountOfSignatures**(const SignedDoc* pSigDoc); **getSignature**(const SignedDoc* pSigDoc, int nIdx); **ddocCertGetSubjectCN**(X509* pCert, DigiDocMemBuf* pMemBuf); **verifySignatureAndNotary**(SignedDoc* pSigDoc, SignatureInfo* pSigInfo, const char* szFileName); **getCountOfSignerRoles**(SignatureInfo* pSigInfo, int nCertified); **getSignerRole**(SignatureInfo* pSigInfo, int nCertified, int nIdx); **ddocSigInfo_GetSignersCert**(const SignatureInfo* pSigInfo); **getNotaryWithSigId**(const SignedDoc* pSigDoc, const char* sigId); **ddocNotInfo_GetResponderId**(const NotaryInfo* pNotary); **ReadCertSerialNumber**(char* szSerial, int nMaxLen, X509 *x509); **ddocCertGetIssuerDN**(X509* pCert, DigiDocMemBuf* pMemBuf); **ddocCertGetSubjectDN**(X509* pCert, DigiDocMemBuf* pMemBuf); **getCertNotBefore**(const SignedDoc* pSigDoc, X509* cert, char* timestamp, int len); **getCertNotAfter**(const SignedDoc* pSigDoc, X509* cert, char*

CDigiDoc utility's command	Called CDigiDoc API method(s)
	timestamp, int len); **readCertPolicies**(X509* pX509, PolicyIdentifier** pPolicies, int* nPols);
-extract <data-file-id> <output-file>	**ddocExtractDataFile**(SignedDoc* pSigDoc, const char* szFileName, const char* szDataFileName, const char* szDocId, const char* szCharset);
-get-confirmation <signature-id>	**getSignatureWithId**(const SignedDoc* pSigDoc, const char* id); **notarizeSignature**(SignedDoc* pSigDoc, SignatureInfo* pSigInfo);
-mid-sign <phone-no> <per-code> [[<country>(EE)] [<lang>(EST)] [<service>(Testing)] [<manifest>] [<city> <state> <zip>]]	**ConfigItem_lookup_int**(const char* key, int defValue); **ConfigItem_lookup**(const char* key); **ddsSign**(SignedDoc* pSigDoc, const char* szIdCode, const char* szPhoneNo, const char* szLang, const char* szServiceName, const char* manifest, const char* city, const char* state, const char* zip, const char* country, char* url, char* proxyHost, char* proxyPort, long* pSesscode, char* szChallenge, int nChalLen); **ddsGetStatus**(SignedDoc* pSigDoc, long lSesscode, char* url, char* proxyHost, char* proxyPort, int* pStatus);
-denc-list <input-file>	**dencSaxReadEncryptedData**(DEncEncryptedData** ppEncData, const char* szFileName); **dencMetaInfo_GetLibVersion**(DEncEncryptedData* pEncData, char* szLibrary, int nLibLen, char* szVersion, int nVerLen); **dencMetaInfo_GetFormatVersion**(DEncEncryptedData* pEncData, char* szFormat, int nFormat, char* szVersion, int nVersion);
-encrecv <certificate-file>	**dencEncryptedData_new**(DEncEncryptedData** pEncData, const char* szXmlNs, const char* szEncMethod, const char* szId, const char* szType, const char* szMimeType); **dencMetaInfo_SetLibVersion**(DEncEncryptedData* pEncData); **dencMetaInfo_SetFormatVersion**(DEncEncryptedData* pEncData); **ReadCertificate**(X509 **x509, const char *szCertfile); **ddocCertGetSubjectCN**(X509* pCert, DigiDocMemBuf* pMemBuf); **dencEncryptedKey_new**(DEncEncryptedData* pEncData, DEncEncryptedKey** pEncKey, X509* pCert, const char* szEncMethod, const char* szId, const char* szRecipient, const char* szKeyName, const char* szCarriedKeyName);
-encrypt-sk <input-file>	**ConfigItem_lookup_int**(const char* key, int defValue); **dencEncryptedData_new**(DEncEncryptedData** pEncData, const char* szXmlNs, const char* szEncMethod, const char* szId, const char* szType, const char* szMimeType); **dencMetaInfo_SetLibVersion**(DEncEncryptedData* pEncData); **dencMetaInfo_SetFormatVersion**(DEncEncryptedData* pEncData); **ddocConvertInput**(const char* src, char** dest); **dencEncryptionProperty_new**(DEncEncryptedData* pEncData, DEncEncryptionProperty** ppEncProperty, const char* szId, const char* szTarget, const char* szName, const char* szContent); **SignedDoc_new**(SignedDoc **pSignedDoc, const char* format, const char* version);

CDigiDoc utility's command	Called CDigiDoc API method(s)
	calculateFileSize(const char* szFileName, long* lFileLen); **DataFile_new**(DataFile **newDataFile, SignedDoc* pSigDoc, const char* id, const char* filename, const char* contentType, const char* mime, long size, const byte* digest, int digLen, const char* digType, const char* szCharset); **dencOrigContent_registerDigiDoc**(DEncEncryptedData* pEncData, SignedDoc* pSigDoc); **createSignedDoc**(SignedDoc* pSigDoc, const char* szOldFile, const char* szOutputFile); **ddocReadFile**(const char* szFileName, DigiDocMemBuf* pData); **dencEncryptedData_encryptData**(DEncEncryptedData* pEncData, int nCompressOption);
-encrypt <input-file>	**ConfigItem_lookup_int**(const char* key, int defValue); **dencEncryptedData_new**(DEncEncryptedData** pEncData, const char* szXmlNs, const char* szEncMethod, const char* szId, const char* szType, const char* szMimeType); **dencMetaInfo_SetLibVersion**(DEncEncryptedData* pEncData); **dencMetaInfo_SetFormatVersion**(DEncEncryptedData* pEncData); **dencEncryptionProperty_new**(DEncEncryptedData* EncData, DEncEncryptionProperty** ppEncProperty, const char* szId, const char* szTarget, const char* szName, const char* szContent); **dencEncryptedData_AppendData**(DEncEncryptedData* pEncData, const char* data, int len); **ddocSaxReadSignedDocFromFile**(SignedDoc** ppSigDoc, const char* szFileName, int checkFileDigest, long lMaxDFLen); **dencOrigContent_registerDigiDoc**(DEncEncryptedData* pEncData, SignedDoc* pSigDoc); **dencEncryptedData_encryptData**(DEncEncryptedData* pEncData, int nCompressOption);
-encrypt-file <input-file> <output-file>	**dencEncryptFile**(DEncEncryptedData* pEncData, const char* szInputFileName, const char* szOutputFileName, const char* szMimeType);
-decrypt-sk <output-file> <pin>	Functions of –decrypt command. **utf82unicode**(const char* utf8, char** unicode, int* outlen); **ddocSaxReadSignedDocFromFile**(SignedDoc** ppSigDoc, const char* szFileName, int checkFileDigest, long lMaxDFLen); **getCountOfDataFiles**(const SignedDoc* pSigDoc); **getDataFile**(const SignedDoc* pSigDoc, int nIdx); **ddocExtractDataFile**(SignedDoc* pSigDoc, const char* szFileName, const char* szDataFileName, const char* szDocId, const char* szCharset);
-decrypt <output-file> <pin>	**dencSaxReadEncryptedData**(DEncEncryptedData** ppEncData, const char* szFileName); **dencEncryptedData_findEncryptedKeyByPKCS12**(DEncEncryptedData* pEncData, DEncEncryptedKey** ppEncKey, EVP_PKEY** ppKey, const char* szPkcs12File, const char* szPasswd); **dencEncryptedData_findEncryptedKeyByPKCS11UsingSlot**(DEncEncryptedData* pEncData, DEncEncryptedKey** ppEncKey, int

CDigiDoc utility's command	Called CDigiDoc API method(s)
	nSlot); **dencEncryptedData_decryptWithKey**(DEncEncryptedData* pEncData, DEncEncryptedKey* pEncKey, EVP_PKEY* pKey); **dencEncryptedData_decryptUsingSlot**(DEncEncryptedData* pEncData, DEncEncryptedKey* pEncKey, const char* pin, int nSlot);
-decrypt-file <input-file> <output-file> <pin>	**dencSaxReadDecryptFile**(const char* szInputFileName, const char* szOutputFileName, const char* szPin, const char* szPkcs12File);

Appendix 1: CDigiDoc configuration file

A sample CDigiDoc configuration 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

```
#-------------------------------------------------
# DigiDoc library global configuration file
#-------------------------------------------------

# PKCS#11 module settings - change this according to your signature device!!!
DIGIDOC_DEFAULT_DRIVER   =       1
DIGIDOC_DRIVERS  =       1
DIGIDOC_DRIVER_1_NAME    =       OpenSC
DIGIDOC_DRIVER_1_DESC    =       OpenSC projects PKCS#11 driver
DIGIDOC_DRIVER_1_FILE    =       opensc-pkcs11.dll
# for Linux: DIGIDOC_DRIVER_1_FILE = opensc-pkcs11.so

# Digital signing settings
# Identifier of the signer's private key's slot on an identity token.
DIGIDOC_SIGNATURE_SLOT   =       1

# Default OCSP responder URL
DIGIDOC_OCSP_URL         =       http://ocsp.sk.ee

# Sign OCSP requests or not. Depends on your responder
# Set this parameter value to "true" if OCSP requests need to be signed
SIGN_OCSP        =       false
# The PKCS#12 file used to sign OCSP requests
# DIGIDOC_PKCS_FILE = <your-pkcs12-file-name>
# Password for this key
# DIGIDOC_PKCS_PASSWD = <your-pkcs12-passwd>

# Your HTTP proxy if necessary
USE_PROXY        =       false
# DIGIDOC_PROXY_HOST = <your-proxy-hostname>
# DIGIDOC_PROXY_PORT = <proxy-port>
# DIGIDOC_PROXY_USER = <proxy-username>
# DIGIDOC_PROXY_PASS = <proxy-password>

# Signature verification settings
CHECK_OCSP_NONCE         =       false
CHECK_SIGNATURE_VALUE_ASN1=      true

# CA certificates
CA_CERT_PATH     =       C:\Program Files\Estonian ID Card Development\libdigidoc\certs
CA_CERTS         =       16

CA_CERT_1        =       JUUR-SK.crt
CA_CERT_1_CN     =       Juur-SK
CA_CERT_2        =       ESTEID-SK.crt
CA_CERT_2_CN     =       ESTEID-SK
CA_CERT_3        =       ESTEID-SK 2007.crt
CA_CERT_3_CN     =       ESTEID-SK 2007
```

```
CA_CERT_4       =       KLASS3-SK.crt
CA_CERT_4_CN    =       KLASS3-SK
CA_CERT_5       =       KLASS3-SK 2010.crt
CA_CERT_5_CN    =       KLASS3-SK 2010
CA_CERT_6       =       EID-SK.crt
CA_CERT_6_CN    =       EID-SK
CA_CERT_7       =       EID-SK 2007.crt
CA_CERT_7_CN    =       EID-SK 2007

CA_CERT_8       =       EECCRCA.crt
CA_CERT_8_CN    =       EE Certification Centre Root CA
CA_CERT_9       =       ESTEID-SK 2011.crt
CA_CERT_9_CN    =       ESTEID-SK 2011
CA_CERT_10      =       EID-SK 2011.crt
CA_CERT_10_CN   =       EID-SK 2011

CA_CERT_11      =       TEST Juur-SK.crt
CA_CERT_11_CN   =       TEST Juur-SK
CA_CERT_12      =       TEST-SK.crt
CA_CERT_12_CN   =       TEST-SK

CA_CERT_13      =       TEST EECCRCA.crt
CA_CERT_13_CN   =       TEST of EE Certification Centre Root CA
CA_CERT_14      =       TEST ESTEID-SK 2011.crt
CA_CERT_14_CN   =       TEST of ESTEID-SK 2011
CA_CERT_15      =       TEST EID-SK 2011.crt
CA_CERT_15_CN   =       TEST of EID-SK 2011
CA_CERT_16      =       TEST KLASS3 2010.crt
CA_CERT_16_CN   =       TEST of KLASS3-SK 2010

# OCSP responder certificates
# Note: if you add or remove some of these certificates, update the following number,
# also pay attention to proper naming
DIGIDOC_OCSP_RESPONDER_CERTS    =       18

DIGIDOC_OCSP_RESPONDER_CERT_1   =       TEST-SK OCSP 2005.crt
DIGIDOC_OCSP_RESPONDER_CERT_1_CN =      TEST-SK OCSP RESPONDER 2005
DIGIDOC_OCSP_RESPONDER_CERT_1_CA =      TEST-SK
DIGIDOC_OCSP_RESPONDER_CERT_1_URL       =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_OCSP_RESPONDER_CERT_2   =       KLASS3-SK OCSP 2009.crt
DIGIDOC_OCSP_RESPONDER_CERT_2_CN =      KLASS3-SK OCSP RESPONDER 2009
DIGIDOC_OCSP_RESPONDER_CERT_2_CA =      KLASS3-SK

DIGIDOC_OCSP_RESPONDER_CERT_3   =       ESTEID-SK OCSP 2005.crt
DIGIDOC_OCSP_RESPONDER_CERT_3_CN =      ESTEID-SK OCSP RESPONDER 2005
DIGIDOC_OCSP_RESPONDER_CERT_3_CA =      ESTEID-SK

DIGIDOC_OCSP_RESPONDER_CERT_4   =       ESTEID-SK 2007 OCSP.crt
DIGIDOC_OCSP_RESPONDER_CERT_4_CN =      ESTEID-SK 2007 OCSP RESPONDER
DIGIDOC_OCSP_RESPONDER_CERT_4_CA =      ESTEID-SK 2007

DIGIDOC_OCSP_RESPONDER_CERT_5   =       EID-SK 2007 OCSP.crt
DIGIDOC_OCSP_RESPONDER_CERT_5_CN =      EID-SK 2007 OCSP RESPONDER
DIGIDOC_OCSP_RESPONDER_CERT_5_CA =      EID-SK 2007

DIGIDOC_OCSP_RESPONDER_CERT_6   =       EID-SK OCSP 2006.crt
DIGIDOC_OCSP_RESPONDER_CERT_6_1  =      EID-SK OCSP.crt
DIGIDOC_OCSP_RESPONDER_CERT_6_CN =      EID-SK OCSP RESPONDER
DIGIDOC_OCSP_RESPONDER_CERT_6_CA =      EID-SK
```

```
DIGIDOC_OCSP_RESPONDER_CERT_7    =       ESTEID-SK OCSP.crt
DIGIDOC_OCSP_RESPONDER_CERT_7_CN =       ESTEID-SK OCSP RESPONDER
DIGIDOC_OCSP_RESPONDER_CERT_7_CA =       ESTEID-SK

DIGIDOC_OCSP_RESPONDER_CERT_8    =       KLASS3-SK OCSP 2006.crt
DIGIDOC_OCSP_RESPONDER_CERT_8_1  =       KLASS3-SK OCSP.crt
DIGIDOC_OCSP_RESPONDER_CERT_8_CN =       KLASS3-SK OCSP RESPONDER
DIGIDOC_OCSP_RESPONDER_CERT_8_CA =       KLASS3-SK

DIGIDOC_OCSP_RESPONDER_CERT_9    =       EID-SK 2007 OCSP 2010.crt
DIGIDOC_OCSP_RESPONDER_CERT_9_CN =       EID-SK 2007 OCSP RESPONDER 2010
DIGIDOC_OCSP_RESPONDER_CERT_9_CA =       EID-SK 2007

DIGIDOC_OCSP_RESPONDER_CERT_10   =       ESTEID-SK 2007 OCSP 2010.crt
DIGIDOC_OCSP_RESPONDER_CERT_10_CN        =       ESTEID-SK 2007 OCSP RESPONDER 2010
DIGIDOC_OCSP_RESPONDER_CERT_10_CA        =       ESTEID-SK 2007

DIGIDOC_OCSP_RESPONDER_CERT_11   =       KLASS3-SK 2010 OCSP.crt
DIGIDOC_OCSP_RESPONDER_CERT_11_CN        =       KLASS3-SK 2010 OCSP RESPONDER
DIGIDOC_OCSP_RESPONDER_CERT_11_CA        =       KLASS3-SK 2010

DIGIDOC_OCSP_RESPONDER_CERT_12   =       SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_12_CN        =       SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_12_CA        =       EE Certification Centre Root CA

DIGIDOC_OCSP_RESPONDER_CERT_13   =       SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_13_CN        =       SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_13_CA        =       ESTEID-SK 2011

DIGIDOC_OCSP_RESPONDER_CERT_14   =       SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_14_CN        =       SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_14_CA        =       EID-SK 2011

DIGIDOC_OCSP_RESPONDER_CERT_15   =       TEST SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_15_CN        =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_15_CA        =       TEST of EE Certification Centre Root CA
DIGIDOC_OCSP_RESPONDER_CERT_15_URL       =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_OCSP_RESPONDER_CERT_16   =       TEST SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_16_CN        =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_16_CA        =       TEST of ESTEID-SK 2011
DIGIDOC_OCSP_RESPONDER_CERT_16_URL       =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_OCSP_RESPONDER_CERT_17   =       TEST SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_17_CN        =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_17_CA        =       TEST of EID-SK 2011
DIGIDOC_OCSP_RESPONDER_CERT_17_URL       =       http://www.openxades.org/cgi-bin/ocsp.cgi

DIGIDOC_OCSP_RESPONDER_CERT_18   =       TEST SK OCSP 2011.crt
DIGIDOC_OCSP_RESPONDER_CERT_18_CN        =       TEST of SK OCSP RESPONDER 2011
DIGIDOC_OCSP_RESPONDER_CERT_18_CA        =       TEST of KLASS3-SK 2010
DIGIDOC_OCSP_RESPONDER_CERT_18_URL       =       http://www.openxades.org/cgi-bin/ocsp.cgi

# 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

# Debugging settings
```

```
# Specifies the amount of information printed out. Possible value range: 0-9
# DEBUG_LEVEL    =       3
# Note that the directory where you want to store the output file has to exist before
# debugging, otherwise the file is not created.
# DEBUG_FILE     =       <your-debugging-log-file>
```