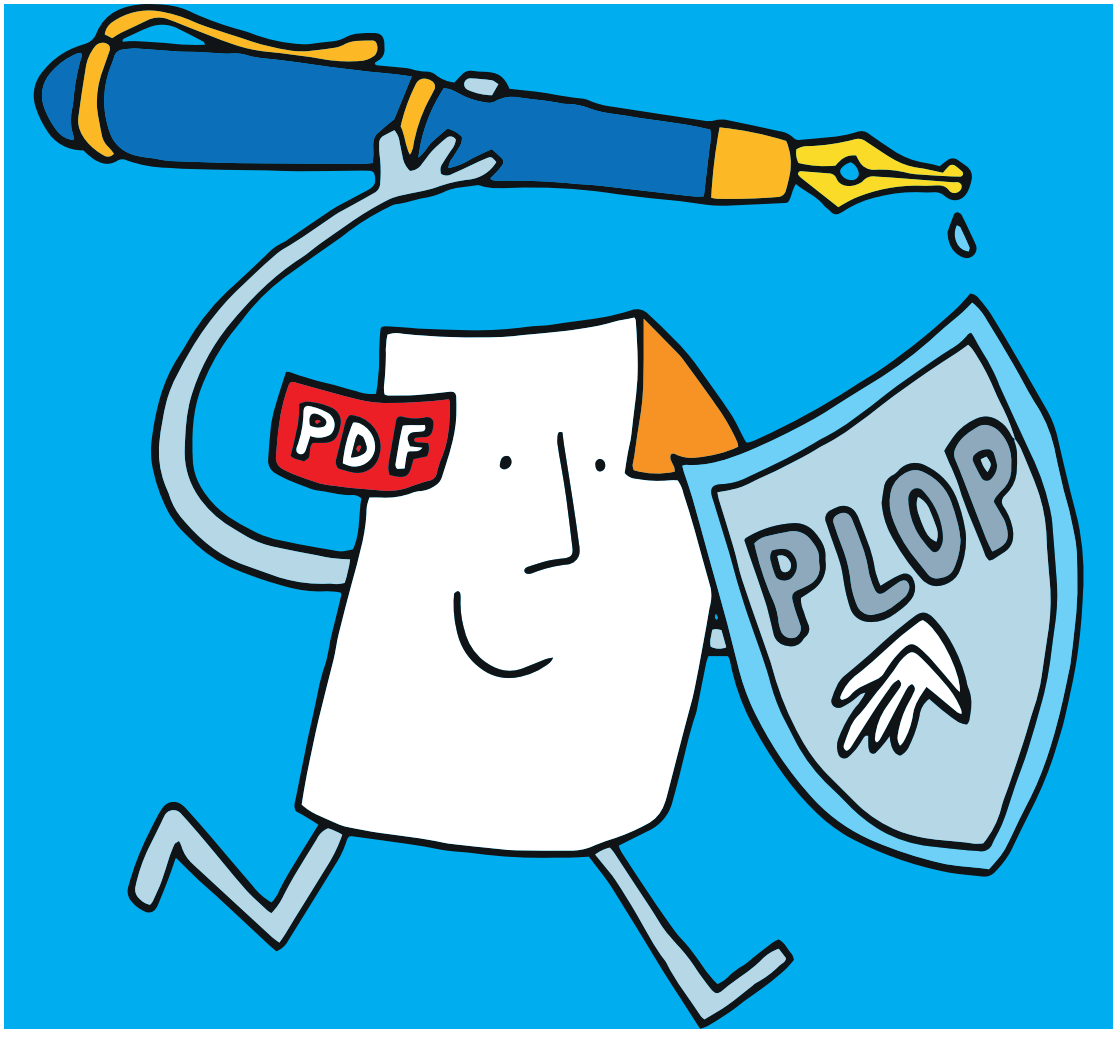
PDFlib GmbH

PLOP and PLOP DS

Version 5.5

PDF Linearization, Optimization, Protection and Digital Signature



PDFlib GmbH
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0

sales@pdflib.com
PDFlib_support@apryse.com (please include your license number)

Contents

0 First Steps with PLOP and PLOP DS

0.1 Installing the Software

PLOP and PLOP DS are delivered as a combined installer package for Windows systems, and as a combined compressed archive for all other supported operating systems. The installer and the archive contain the PLOP/PLOP DS command-line tool and the PLOP/PLOP DS library, plus documentation and examples. After installing or unpacking the package the following steps are recommended:

- An introduction to the features is available in Chapter 1, »PLOP Features«, page 17, and Chapter 2, »PLOP DS Features (Digital Signature)«, page 31.
- Users of the PLOP/PLOP DS command-line tool can use the executable right away. The available options are discussed in Section 3.1, »PLOP and PLOP DS Command-line Options«, page 39, and are also displayed when you execute the PLOP command-line tool without any options.
- Users of the PLOP/PLOP DS library/component should read one of the sections in Chapter 4, »PLOP and PLOP DS Library Language Bindings«, page 45, corresponding to their environment of choice, and review the installed examples. On Windows the PLOP and PLOP DS programming examples are accessible via the Start menu (for .NET) or in the installation directory (for other language bindings).

If you obtained a commercial PLOP or PLOP DS license you must apply your license key according to the next page.

Restrictions of the evaluation version. The PLOP/PLOP DS command-line tool and library can be used as fully functional evaluation versions even without a commercial license. Unlicensed versions of PLOP or PLOP DS must not be used for production purposes, but only for evaluating the product. Deploying the software in a production environment requires a valid license.

Unless a valid license key is applied, PLOP includes the text *unlicensed* in the output document's metadata and inserts an extra front page at the beginning of the document. In order to facilitate testing, no front page is created if one of the following conditions are true:

- Encryption with the fixed password strings *demo* or *DEMO* (options *userpassword* and *masterpassword)*.
- Applying a signature with a digital ID where the common name (*CN)* in the *subject* field contains *demo* or *DEMO;* suitable digital IDs for testing are included in the PLOP DS package.
- Protecting a document with certificate security with recipient certificates where the common name *(CN)* in the *subject* field contains *demo* or *DEMO*; suitable certificates for testing are included in the PLOP DS package.

In some situations insertion of the front page may result in PDF output which no longer conforms to PDF/A, PDF/UA, PDF/VT or PDF/X even if the input conforms to one of these standards. The non-conformance is specific to the front page and is no longer an issue once a valid license key is applied.

pCOS functions are restricted to small documents (less than 10 pages and less than 1 MB) in evaluation mode.

For each document handle retrieved from *plop.open_document( )*, only a single call to *plop.create_document( )* call is allowed in evaluation mode.

0.2 Applying the PLOP/PLOP DS License Key

Using PLOP/PLOP DS for production purposes requires a valid license key. Once you purchased a license you must apply your license key in order to get rid of the extra front page and enable the use of arbitrary passwords. There are several methods for applying the license key; choose one of the methods detailed below.

If the *frontpage* option for *set_option( )* is *false,* an exception is thrown instead of creating the front page when no valid license key could be found.

Note PLOP/PLOP DS license keys are platform-dependent, and can only be used on the platform for which they have been purchased. While a PLOP DS license key activates all features of PLOP, a PLOP license key does not activate the signature features which are only available in PLOP DS.

Note A PLOP or PLOP DS license also covers the pCOS command-line tool which is included in all PLOP packages.

Windows installer. Windows users can enter the license key when they install PLOP/PLOP DS using the supplied installer. This is the recommended method on Windows. If you do not have write access to the registry or cannot use the installer refer to one of the alternate methods below.

Applying a license key with an API call at runtime. Add a line to your script or program which sets the license key at runtime. The *license* parameter must be set immediately after instantiating the PLOP object (i.e., after *new( )* or equivalent call). The exact syntax depends on your programming language:

- In C++, Java, .NET/C#, Python and Ruby:

```
plop.set_option("license=...your license key...")
```

- In C:

```
PLOP_set_option(p, "license=...your license key...");
```

- In Perl and PHP:

```
$plop->set_option("license=...your license key...")
```

Working with a license file. As an alternative to supplying the license key with a runtime call, you can enter the license key in a text file according to the following format (you can use the license file template *licensekeys.txt* which is contained in all PLOP distributions). Lines beginning with a '#' character contain comments and will be ignored; the second line contains version information for the license file itself:

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
PLOP        5.5             ...your license key...
```

The license file may contain license keys for multiple PDFlib GmbH products on separate lines. It may also contain license keys for multiple platforms so that the same license file can be shared among platforms. License files can be configured in the following ways:

- A file called *licensekeys.txt* is searched in all default locations (see »Default file search paths«, page 10).
- You can specify the *licensefile* parameter with the *set_option( )* API method:

```
plop.set_option("licensefile={/path/to/licensekeys.txt}");
```

- Use the *--plopopt* option of the PLOP or pCOS command-line tool and supply the *licensefile* option with the name of a license file:

  ```
  plop --plopopt "licensefile /path/to/your/licensekeys.txt" ...
  pcos --plopopt "licensefile /path/to/your/licensekeys.txt" ...
  ```

  If the path name contains space characters you must enclose the path with braces:

  ```
  plop --plopopt "licensefile {/path/to/your/license file.txt}" ...
  pcos --plopopt "licensefile {/path/to/your/license file.txt}" ...
  ```

- You can set an environment (shell) variable which points to a license file. On Windows use the system control panel and choose *System, Advanced, Environment Variables*.; on Unix apply a command similar to the following:

  ```
  export PDFLIBLICENSEFILE=/path/to/licensekeys.txt
  ```

License keys in the registry. On Windows you can also enter the name of the license file in the following registry value:

```
HKLM\SOFTWARE\PDFlib\PDFLIBLICENSEFILE
```

As another alternative you can enter the license key directly in one of the following registry values:

```
HKLM\SOFTWARE\PDFlib\PLOP5\license
HKLM\SOFTWARE\PDFlib\PLOP5\5.5\license
```

The installer writes the license key to the last of these entries.

Note *Be careful when manually accessing the registry on 64-bit Windows systems: as usual, 64-bit PLOP binaries work with the 64-bit view of the Windows registry, while 32-bit PDFlib binaries running on a 64-bit system work with the 32-bit view of the registry. If you must add registry keys for a 32-bit product manually, make sure to use the 32-bit version of the* `regedit` *tool. It can be invoked as follows from the* `Start` *dialog:*

```
%systemroot%\syswow64\regedit
```

Default file search paths. On Unix, Linux and macOS systems some directories will be searched for files by default even without specifying any path and directory names. The following directories will be searched:

```
<rootpath>/PDFlib/PLOP/5.5/resource/cmap
<rootpath>/PDFlib/PLOP/5.5/resource/codelist
<rootpath>/PDFlib/PLOP/5.5/resource/glyphlst
<rootpath>/PDFlib/PLOP/5.5/resource/fonts
<rootpath>/PDFlib/PLOP/5.5/resource/icc
<rootpath>/PDFlib/PLOP/5.5
<rootpath>/PDFlib/PLOP
<rootpath>/PDFlib
```

On Unix, Linux, and macOS <*rootpath*> will first be replaced with */usr/local* and then with the HOME directory.

Default file names for license files. By default, the following file name will be searched for in the default search path directories:

```
licensekeys.txt
```

This feature can be used to work with a license file without setting any environment variable or runtime option.

0.3 Roadmap to Documentation and Samples

Mini samples for PLOP. The PLOP distribution contains simple programming examples for all supported language bindings. These demonstrate basic PLOP library programming tasks:

- The *encrypt* sample encrypts an unencrypted PDF document with user and master password.
- The *certsec* sample encrypts a PDF document against one ore more recipient certificates. Sample recipient certificates are included in the package. The package also includes the corresponding digital ID files which are required to decrypt the encrypted documents with PLOP or open them in Acrobat. The password for all digital ID files (e.g. *demo_recipient_1.p12)* is *demo*.
- The *dumper* sample uses the pCOS interface to collect general properties, information about the encryption and signature status of a document as well as document information and XMP metadata.
- The *insertxmp* sample reads XMP metadata from a file, and inserts the XMP in a PDF document. Sample XMP files are supplied for testing.

Mini samples for PLOP DS. The following mini samples are for use with PLOP DS:

- The *sign* sample shows how to apply a digital signature to an existing PDF document.
- The *multisign* sample shows how to apply digital signature to multiple PDF documents and demonstrates session handling for PKCS#11 tokens.
- The *hellosign* sample shows how to dynamically create a document with PDFlib in memory and pass it to PLOP DS, which then applies a digital signature to it. This sample requires the PDFlib product which is not included in the PLOP package. Free evaluation packages for PDFlib are available from our Web site, however.
- The *dynamicsign* sample shows how to dynamically create a signature visualization document with PDFlib, e.g. containing a personalized image of a handwritten signature. This sample also requires the PDFlib product.

The signature samples are prepared to use demo digital IDs which are included in the package. The password for all digital ID files (e.g. *demo_signer_rsa_2048.p12)* is *demo*.

Sample calls of the PLOP command-line tool. The PLOP command-line tool supports various options which are documented in Section 3.1, »PLOP and PLOP DS Command-line Options«, page 39. Several other chapters in this manual also contain sample calls of the PLOP command-line tool.

pCOS Cookbook. The *pCOS Cookbook* is a collection of code fragments for the pCOS interface which is integrated in PLOP and PLOP DS. It is available at the following URL: *www.pdflib.comhttps://www.pdflib.com/pcos-cookbook*.

Details of the pCOS interface are documented in the pCOS Path Reference which is included in the PLOP package.

Sample calls of the pCOS command-line tool. The pCOS command-line tool, which is included in all PLOP packages, is discussed in a separate manual which also contains sample calls.

0.4 Overview of PLOP and PLOP DS

PLOP is available in two flavors: the PLOP base product and the extended version PLOP DS with support for digital signatures.

PLOP features. PLOP supports the following kinds of PDF processing:

- Password security: encrypt a PDF document with a user or master password (or both); remove PDF encryption if you know the document's master password; add or remove permission settings (e.g., printing or text extraction not allowed) if you know the document's master password.
- Certificate security: encrypt a PDF document for one or more recipient certificates; decrypt protected PDF documents with a suitable digital ID. Apply or remove permission settings for certificate security.
- Linearize PDF documents for enhanced viewer experience when retrieving PDF files from a Web server.
- Optimize the size of PDF documents by reducing redundant objects.
- Repair damaged PDF documents.
- Use the integrated pCOS interface to query information about the document's security status (encrypted with user or master password), permission settings, document metadata, and many other properties.
- Insert and retrieve predefined or custom document information entries.
- Insert and retrieve XMP metadata.

PLOP DS features. PLOP DS offers all features of PLOP, plus the ability to apply digital signatures to PDF documents. The signatures support timestamping, long-term validation and PAdES signatures. Section 2.1, »Signature Features in PLOP DS«, page 31, provides a summary of digital signature features in PLOP DS.

Advantages. PDFlib PLOP and PLOP DS offer the following advantages:

- All PLOP and PLOP DS operations are aware of the PDF/A, PDF/UA, PDF/VT and PDF/X standards: if the input conforms to one of these standards, the output is guaranteed to conform to the same standard if possible. If this is not possible (e.g. encryption was requested for PDF/A input) the operation will either be rejected or the standard identification removed.
- PLOP/PLOP DS is a standalone tool which does not require any third-party software for reading, encrypting, signing, or writing PDF.
- PLOP/PLOP DS can be deployed on a server, is fully thread-safe, and has been checked for memory leaks. PLOP has been engineered for heavy server usage, and can be used in Web server environments, for high-volume batch processing, etc.
- PLOP/PLOP DS is available on many platforms and for several programming environments.
- For added flexibility, PLOP/PLOP DS is available both as a command-line tool and a programming library (component) for various development languages.

PLOP/PLOP DS command-line tool or library? PLOP/PLOP DS is available both as a programming library (component) for various development languages, and as a command-line tool for batch operations. Both offer the same feature set, but are suitable for different deployment tasks. Here are some guidelines for choosing among the library and the command-line tool:

- The command-line PLOP/PLOP DS tool is suited for batch processing PDF documents. It doesn't require any programming, but offers powerful command-line options which can be used to integrate it into complex workflows. The PLOP/PLOP DS command-line tool can also be called from environments which do not support the use of the library.
- The PLOP/PLOP DS programming library integrates well into a variety of common development environments, such as .NET, Java (including servlets), PHP, and plain C or C++ application development.

The PLOP/PLOP DS license covers both the command-line tool and the library.

0.5 What's new in PLOP and PLOP DS?

General changes in PLOP 5.1:

- certificate security: encrypt a document against a set of recipients which are identified by their digital certificate
- pCOS interface 11 for retrieving details of documents encrypted with certificate security
- updated language bindings and platform support
- various bug fixes and improvements in the language bindings and kernel

Signature-related changes in PLOP DS 5.1:

- updated timestamping according to RFC 5816 *(SigningCertificateV2/ESSCertIDv2)*
- optimize file size and processing speed for generating signatures
- timing options for OCSP
- support for indirect CRLs
- made CRL retrieval more robust, e.g. unexpected HTTP headers
- workarounds in the PKCS#11 engine to handle behavior of certain token models
- PKCS#11 improvements for multi-threaded signing applications
- support for signing with a Hardware Security Module (HSM)
- create PAdES/CAdES signatures by default
- custom build of PLOP DS for attaching external cryptographic engine for hashing and signing
- bug fixes in PDF handling, e.g. form field names, XMP properties
- added code samples for creating dynamic signature visualization with PDFlib
- new build configuration for attaching external crypto routines via the PKCS#11 interface, but without dynamic loading

Changes in PLOP DS 5.2:

- support PSS encoding scheme for RSA signatures
- certificate security: support OAEP padding scheme for RSA

Changes in PLOP 5.3:

- updated language bindings and platform support
- pCOS interface 12 with support for identifying upcoming ISO standards and improvements in the security model for accessing form fields in restricted mode
- added the pCOS command-line tool
- support PSS encoding scheme for RSA signatures also via PKCS#11
- support for Amazon CloudHSM

Changes in PLOP 5.4:

- many improvements for platforms and language bindings
- updated third-party code to fix potential security vulnerabilities
- support of proxy servers for network access
- don't create a separate PDF update for the DSS since some signature validation tools complained that »Signature doesn't cover whole document«.
- enable transport compression for CRLs and OCSP to accelerate network transmission
- add workaround for misconfigured CAs when downloading CRLs
- more workarounds for malformed PDF and XMP metadata
- support for Alpine Linux

Changes in PLOP 5.5:

- security and performance updates of third-party components
- enhancements in all language bindings and updates for the latest language versions including .NET 6/7, PHP 8.1/8.2, Perl 5.34/5.36 and Ruby 3.1
- added support for arm64/x86_64 bindings on macOS

1 PLOP Features

Note *PLOP DS features for digital signatures are presented in Chapter 2, »PLOP DS Features (Digital Signature)«, page 31.*

1.1 Password Security and Permissions

Encrypting and decrypting PDF documents with passwords as well as permission restrictions are covered in detail in Chapter 5, »Password Security«, page 61. In the current section we provide a summary and some initial examples.

Querying security settings. With the pCOS programming interface you can query various security settings of a PDF document which has been protected with password security. The required function calls and parameters can be seen in the *dumper* mini sample, which is included in all PLOP packages. The pCOS command-line tool can be used to query information from PDF documents without any programming (see Section 1.6, »Query Document Information with pCOS«, page 22, for an example).

Encrypting documents with a password. You can encrypt documents by specifying the *userpassword* or *masterpassword* option (or both) for *create_document( )*. Note that a user password always requires a master password, but not vice versa. Sample code for encrypting PDF documents can be seen in the *encrypt* sample which is included in all PLOP packages. The equivalent options for the PLOP command-line tool are *--user* and *--master*.

Example: encrypt a file with user password *demo* and master password *DEMO*:

```
plop --user demo --master DEMO --outfile encrypted.pdf input.pdf
```

Specify permission restrictions. You can specify the permission restrictions in the *permissions* option of *create_document( )* which supports various keywords (see Table 5.3, page 66). The equivalent option for the PLOP command-line tool is *--permissions*. Note that permission restrictions always require a master password.

Example: encrypt a document with the master password DEMO, and disallow printing the document and copying contents:

```
plop --master DEMO --permissions "noprint nocopy" --outfile encrypted.pdf input.pdf
```

Decrypting password-protected documents. You can decrypt documents by specifying the appropriate user or master password in the *password* option for *open_document( )*. The equivalent option for the PLOP command-line tool is *--password*.

Example: decrypt a single file with the master password *DEMO*. All access restrictions which may have been applied to the input document will be removed (since the output is unencrypted):

```
plop --password DEMO --outfile decrypted.pdf encrypted.pdf
```

More encryption and decryption examples can be found in Section 5.3, »Applying Password Security on the Command-Line«, page 68.

1.2 Certificate Security

Encrypting and decrypting PDF documents with certificates are covered in detail in Chapter 6, »Certificate Security«, page 71. In the current section we provide a summary and some initial examples.

Querying security settings. With the pCOS programming interface you can query various security settings of a PDF document which has been protected with certificate security. The required function calls and parameters can be seen in the *dumper* mini sample, which is included in all PLOP packages. The pCOS command-line tool can be used to query information from PDF documents without any programming (see Section 1.6, »Query Document Information with pCOS«, page 22, for an example).

Encrypting documents with a certificate. You can encrypt documents by specifying the a recipient *certificate* with *add_recipient( )*. Sample code for encrypting PDF documents can be seen in the *certsec* sample which is included in all PLOP packages. The equivalent option for the PLOP command-line tool is *--recipient*.

Example: encrypt a file with a certificate:

```
plop --recipient "certificate={filename=demo_recipient_1.pem}" ↵
       --outfile encrypted.pdf input.pdf
```

Specify permission restrictions. You can specify permission restrictions for a recipient in the *permissions* option of *add_recipient( )*.

Example: Encrypt a document for a recipient and restrict its permissions so that printing and copying are not allowed:

```
plop --recipient "certificate={filename=demo_recipient_1.pem permissions={noprint ↵
       nocopy}}" --outfile encrypted.pdf input.pdf
```

Decrypting documents protected with certificate security. You can decrypt documents by specifying an appropriate recipient ID in the *digitalid* option of *open_document( )*. The equivalent option for the PLOP command-line tool is *--inputopt*.

Example: decrypt a single file with a digital id which is available in a password-protected PKCS#12 file:

```
plop --inputopt "digitalid={filename=demo_recipient_1.p12} password=demo" ↵
     --outfile decrypted.pdf encrypted.pdf
```

More encryption and decryption examples can be found in Section 6.5, »Applying Certificate Security on the Command-Line«, page 84.

1.3 Web-Optimized (Linearized) PDF

PLOP can apply a process called linearization to PDF documents. The resulting property is called *Fast Web View* in Acrobat. Linearization reorganizes the objects within a PDF file and adds supplemental information which can be used for faster access.

While non-linearized PDFs must be fully transferred to the client, a Web server can transfer linearized PDF documents one page at a time using a process called byteserving. It allows Acrobat (running as a browser plugin) to retrieve individual parts of a PDF document separately. The result is that the first page of the document will be presented to the user without having to wait for the full document to download from the server. This provides enhanced user experience.

Note that the Web server streams PDF data to the browser, not PLOP. Instead, PLOP prepares the PDF files for byteserving. All of the following requirements must be met in order to take advantage of byteserving PDFs:

- The PDF document must be linearized, which can be achieved with PLOP. Linearization can be applied along with encryption or decryption in a single run. In Acrobat you can check whether a file is linearized by looking at its document properties (»Fast Web View: yes«).
- The user must use Acrobat as a Browser plugin, and have page-at-a-time download enabled in the PDF viewer (Acrobat DC: *Edit, Preferences, Internet, Allow fast web view)*. This is enabled by default.

The larger a PDF file (measured in pages or MB), the more it will benefit from linearization when delivered over the Web.

Linearization and encryption/decryption can be applied in combination. However, in order to linearize a protected file you must provide the proper master password (see Table 5.2).

Linearization and file size. Since linearization aims at improving the Web-based display of large PDF documents it doesn't make much sense for single-page documents (although this is possible). However, due to a bug in Acrobat small linearized documents are not always treated as linearized. For example, Acrobat regards all documents which are smaller than 4KB as non-linearized.

Acrobat also doesn't regard PDF documents larger than 2 GB as linearized.

Linearizing PDF documents with PLOP. You can enable the linearization step with the *linearize* option for *create_document( )*.

The equivalent option for the PLOP command-line tool is *--webopt*. Example: linearize all PDF documents in a directory (assuming these do not require any password), and copy the resulting files to the target directory *output*. Verbosity level 2 prints the names of all input and output files as they are processed:

```
plop --verbose 2 --webopt --targetdir output *.pdf
```

1.4 Optimization (Size Reduction)

While processing PDF documents PLOP can apply file optimization in addition to other operations:

- PLOP detects multiple instances of identical data, and removes all instances but one. This is mostly relevant for fonts and images, but may affect other data types as well, e.g. ICC profiles or even complete pages with identical content. An embedded font or image is removed if another font or image contains the exact same data; all references to the removed data are replaced with references to the remaining instance of the font or image. For example, if a document has been assembled from several PDFs containing parts of a document and all of these parts contain the same embedded font, the resulting combined PDF may carry excess font data. PLOP reduces the redundant font data and keeps only one instance of the font.
- Unused objects are removed from the PDF file in a process known as *garbage collection*. In some cases (when the *Save* menu item in Acrobat has been used, as opposed to *Save As...*) Acrobat appends changes to a file while retaining the previous state of the document. PLOP removes all objects related to older versions of the document.

PLOP never applies any optimization which would result in loss of information (e.g. unembedding fonts, downsampling images). All relevant information for viewing or printing the document in the exact same quality of the input is retained in the output.

Since only a small fraction of today's PDF documents suffers from redundant objects the optimization step is disabled by default.

Optimizing PDF documents with PLOP. You can enable the optimization step with the *optimize=all* option for *create_document( )* or the *--outputopt* option of the PLOP command-line tool.

Example: optimize a document with the PLOP command-line tool:

```
plop --outputopt optimize=all --outfile optimized.pdf input.pdf
```

Removing XMP metadata with PLOP. Some applications create PDF output with large amounts of XMP metadata which are not required in all situations. There are extreme cases where XMP metadata accounts for the vast majority of the total PDF file size. In these cases you can remove unwanted XMP document metadata with PLOP as follows:

```
plop --inputopt xmppolicy=remove --outfile output.pdf input.pdf
```

This may substantially reduce the PDF file size at the expense of detailed metadata. Note that standard identifiers (e.g. for PDF/A) in the XMP are lost.

1.5 Repair Mode for Damaged PDF

PLOP implements a repair mode for damaged PDF so that even certain kinds of damaged documents can be processed. However, in rare cases a damaged PDF document may be rejected if PLOP is unable to repair it.

Repairing PDF documents with PLOP. The repair mode is activated automatically when PLOP encounters damaged input. However, using the *repair=force* option of *open_document( )* you can enforce the repair mode even if no problems occurred when opening the document. The equivalent option for the PLOP command-line tool is *--inputopt repair=force*. You can disable the repair mode with *repair=none*.

Example: force reconstruction of a document with the PLOP command-line tool:

```
plop --inputopt repair=force --outfile repaired.pdf damaged.pdf
```

Invalid XMP metadata. PLOP repairs certain kinds of problems in XMP metadata. However, some problems cannot be repaired. For example XML parsing errors caused by XMP metadata always imply that the XMP is unusable. PLOP provides the *xmppolicy* option for controlling the processing behavior when invalid XMP is encountered. See »Dealing with invalid XMP metadata«, page 26, for more details.

1.6 Query Document Information with pCOS

With the pCOS interface, which is integrated in the PLOP library, you can query various properties of a PDF document. The pCOS interface is covered in detail in the pCOS Path Reference. Sample code for querying document information with pCOS can be seen in the *dumper* mini sample which is included in all PLOP packages.

Querying general document information with the pCOS command-line tool. The pCOS command-line tool displays general encryption information, font names and other information about a PDF document. More examples and a description of all supported options are available in the pCOS command-line tool manual. Sample output:

```
pcos PLOP-manual.pdf
```

This program call results in output similar to the following:

```
          File name: PLOP-manual.pdf
          File size: 1166699
        PDF version: 1.7
         Encryption: No encryption
          Master pw: false
            User pw: false
             nocopy: false (copying is allowed)
           nomodify: false (adding form fields and other changes is allowed)
           noannots: false (adding or changing comments or form fields is allowed)
         noassemble: false (insert/delete/rotate pages, creating bookmarks is allowed)
            noforms: false (filling form fields is allowed)
      noaccessible: false (extracting text or graphics for accessibility is allowed)
      nohiresprint: false (high-resolution printing is allowed)
     plainmetadata: true  (metadata is not encrypted)
        Linearized: true
       PDF/X status: none
       PDF/A status: none
      PDF/UA status: none
      PDF/VT status: none
         Tagged PDF: false
         Signatures: 0
     Reader-enabled: false

       No. of pages: 172
       No. of fonts: 12
           embedded TrueType font PDFlibLogo-Regular
           embedded Type 1 CFF font ThesisAntiqua-Bold
           embedded Type 1 CFF font TheSans-Italic
           ...
           embedded Type 1 CFF font ThesisAntiqua-Normal
           embedded Type 1 CFF font TheSansMonoCondensed-Plain

             Author: 'PDFlib GmbH'
       CreationDate: 'D:20160420105759Z'
            Creator: 'FrameMaker 11.0.2'
            ModDate: 'D:20160420112723+02'00''
           Producer: 'Acrobat Distiller 11.0 (Windows)'
            Subject: 'PDFlib PLOP and PLOP DS: PDF Linearization, Optimization,
Protection, Digital Signature'
              Title: 'PDFlib PLOP and PLOP DS Manual'
```

```
 XMP meta data: is present
Encr. attachm.: no
```

Querying specific information with the pCOS command-line tool. The pCOS command-line tool can also be used to retrieve the value of a particular pCOS path from a document.

The following command lists all annotations (links and other types) with their Subtype, destination within the document, the target URL, and the link rectangle coordinates on the page. Double quotes must surround the list of annotation keys since they must be supplied as a single argument to the program:

```
pcos --extended annotation "Subtype destpage A/URI Rect" file.pdf
```

The following command prints the number of annotations on the first page:

```
pcos --pcospath "length:pages[0]/Annots" file.pdf
```

More examples can be found in the pCOS command-line tool manual.

1.7 Inserting and Reading Document Info Entries

PDF supports two kinds of document metadata which contain general information about a document: document info entries and XMP metadata.

Document info entries are keys with associated strings that hold some unstructured information. The predefined info keys *Subject, Title, Author*, and *Keywords* are commonly used, but arbitrary custom keys can be defined for specific purposes. Document information entries are considered the old and simple kind of PDF metadata, but are deprecated in PDF 2.0.

With PLOP you can add new document information entries or replace the values of existing info entries. Both predefined or custom entries can be set. If the input document contains XMP document metadata, standard and custom info entries are automatically synchronized to XMP in order to keep the metadata consistent.

Inserting document info entries with PLOP. You can set document info entries with the *docinfo* option for *create_document( )*.

Example: specify the predefined document info entry *Subject* and the custom info entry *Department;* note the braces around *Product Manual* to protect the space character:

```
docinfo={Department Techdoc Subject {Product Manual}}
```

This option can be supplied to the PLOP command-line tool via the *--outputopt* option as follows:

```
plop --outputopt "docinfo={Department Techdoc  Subject {Product Manual}}" ↵
        --outfile output.pdf input.pdf
```

In PDF 2.0 the document info dictionary is deprecated. It is therefore only emitted if the *emitdocinfo* option in *create_document( )* is *true*. Even if no document info dictionary is created, document info entries are still synchronized to XMP.

Reading document info entries with PLOP. With the pCOS programming interface, which is integrated in the PLOP library, you can read document information entries (keys and values) from a PDF document. The required function calls and parameters can be seen in the *dumper* mini sample, which is included in all PLOP packages.

The pCOS command-line tool can be used to query information from PDF documents without any programming (see Section 1.6, »Query Document Information with pCOS«, page 22, for an example).

Document info entries in PDF/A. Keep in mind that the PDF/A standard mandates special handling for document info entries:

- PDF/A-1: the standard document info entries *Title, Author, Subject, Keywords, Creator, Producer, CreationDate, ModDate* must be synchronized in the document XMP metadata. PLOP automatically provides this synchronization.
- PDF/A-2/3: document information entries may be present, but must be ignored by PDF/A-conforming readers. If they are present, they should be synchronized with document XMP which is done automatically by PLOP as in the PDF/A-1 case.

1.8 Inserting, Reading, or Removing XMP Metadata

XMP *(Extensible Metadata Platform)* is an XML framework with many predefined properties. As the name implies, XMP can be extended to satisfy specific requirements using custom extension schemas. XMP is much more powerful than document information entries, and is required in PDF/A and various other standards. Many industry groups have published standards based on XMP for various vertical applications, e.g. digital imaging or prepress data exchange.

You can find more detailed information on XMP as well as links to other resources on the PDFlib Web site.

With PLOP you can insert XMP metadata in PDF documents or read XMP from PDF. Inserted XMP is validated to make sure that correct output is created. If the input document conforms to the PDF/A standard, user-supplied XMP must conform to the XMP rules set forth in PDF/A. These rules (including XMP extension schema validation) are checked by PLOP to make sure that PDF/A input plus user-supplied XMP will result in conforming PDF/A output.

XMP insertion with PLOP can be used in the following and many other situations (the names of sample XMP files in the PLOP distribution are provided in parenthesis):

- Add XMP metadata to PDF/A documents, including support for XMP extension schemas as defined in the PDF/A standard *(machine_pdfa1.xmp)*.
- Add XMP metadata describing the scan process for digitized legacy documents *(engineering.xmp)*.
- Add XMP metadata according to the Ghent Workgroup (GWG) Ad Ticket scheme, *(gwg_ad_ticket.xmp)*. For more details see *www.gwg.org/download/job-tickets/*
- Add company-specific XMP metadata *(acme.xmp)*.

Inserting XMP metadata with PLOP. In order to insert metadata you must create a file which contains valid XMP metadata in UTF-8 format. You can insert XMP with the *metadata* option for *create_document( )*, which supports several suboptions. Sample code for inserting XMP in PDF documents is available in the *insertxmp* mini sample, which is included in all PLOP packages.

Example: insert XMP metadata from a file called *gwg_ad_ticket.xmp*, where the XMP is validated against the XMP 2004 standard:

```
plop --outputopt "metadata={filename=gwg_ad_ticket.xmp validate=xmp2004}" ↵
        --outfile output.pdf input.pdf
```

Synchronizing document info entries to XMP. Document info entries inserted with the *docinfo* option for *create_document( )* are automatically synchronized to XMP. This provides a convenient way of creating XMP without having to deal with XMP details.

Standard document info entries are mirrored in the corresponding standard XMP properties. Custom document info entries are mirrored in the *pdfx* XMP schema (the schema name is derived from PDF Extension and is unrelated to the PDF/X standard). Note that custom document info entries are not mirrored to XMP in PDF/A mode since PDF/A requires an extension schema description for custom properties.

Reading XMP metadata with PLOP. With the pCOS programming interface, which is integrated in the PLOP library, you can extract XMP metadata from a PDF document. The required function calls and parameters can be seen in the *dumper* mini sample.

Note that the *dumper* sample does not actually print the XMP metadata, but simply reports the size of the XMP found in the document.

XMP metadata can also be extracted with the pCOS command-line tool.

Removing XMP metadata with PLOP. In some situations you may want to remove XMP metadata, e.g. because it no longer matches the actual document contents. This can be achieved with PLOP as follows:

```
plop --inputopt xmppolicy=remove --outfile output.pdf input.pdf
```

Note that standard identifiers (e.g. for PDF/A) are lost when removing XMP metadata.

Dealing with invalid XMP metadata. PDF documents sometime contain invalid XMP metadata which is either invalid on the XML level or the XMP/RDF level. PLOP will by default reject such documents and stop processing. In order to provide more fine-grain control for such input documents the *xmppolicy* option for *open_document( )* can be used to distinguish the following cases:

- *xmppolicy=rejectinvalid:* by default, invalid XMP prevents PLOP from generating PDF output.
- *xmppolicy=ignoreinvalid:* ignore invalid XMP and include the text of the XML parsing error message in the generated output XMP as a debugging aid. Note that no PDF/A or PDF/X-3/4/5 output can be created with this option.
- *xmppolicy=remove:* remove input XMP. This may be useful to delete unwanted metadata.

For example, if you don't want invalid XMP metadata to disrupt batch processing of documents you can ignore problems caused by invalid XMP in the input document:

```
plop --inputopt "xmppolicy=ignoreinvalid" --outfile output.pdf input.pdf
```

1.9 PLOP Processing Details

Acceptable input documents. PLOP accepts the following PDF flavors:

- PDF 1.6 (Acrobat 7) and all older versions
- PDF 1.7 (Acrobat 8), technically identical to ISO 32000-1
- PDF 1.7 Adobe extension level 3 (Acrobat 9)
- PDF 1.7 Adobe extension level 8 (Acrobat X and above)
- PDF 2.0 according to ISO 32000-2, including dated revision

Depending on the desired operation a password may be required for encrypted documents. PLOP attempts to repair various kinds of damaged PDF documents.

PDF version. The PDF version of the generated output document is never lower than the PDF version number of the input document, but it may be forced to a higher number. PLOP uses the PDF version of the input document, modified according to the following rules:

- In PDF/A-1 and PDF/X mode the PDF version is kept unchanged; in PDF/A-2/3 mode PDF 1.7 is generated.
- Otherwise the PDF output version is at least PDF 1.6.
- Password security (option *masterpassword)* increases the PDF version to PDF 1.7ext3 for encryption algorithm 4 and to PDF 1.7ext8 for encryption algorithm 11 (see »Encryption algorithm and key length for password security«, page 65).
- Certificate security (function *add_recipient( ))* increases the PDF version to PDF 1.6 for pCOS algorithm 6, and to PDF 1.7ext3 for pCOS algorithm 10 (see »PDF encryption algorithm and key length«, page 80).
- Some signature features increase the PDF version to PDF 1.7ext8 (see Table 7.1).

Standard conformance. PLOP processing conforms to several PDF standards. If the input conforms to one of the following standards, the output generated by PLOP is guaranteed to conform to the same standard:

- PDF/A-1/2/3: all flavors
- PDF/X-3/4/5 and PDF/VT-1: all flavors
- PDF/UA-1

Note that some PLOP operations (most importantly encryption) are not compatible with certain standards. In this situation the *sacrifice* option can be used to set priorities (see below).

Sacrificing certain properties of the input PDF. Conflicts can arise between several PDF document properties and certain PLOP actions. For example, PDF/A documents are not allowed to use encryption. What should PLOP do when encryption is requested for a PDF/A document? By default PLOP refuses the operation and throws an exception. However, you can use the option *sacrifice* for *create_document( )* or the *--outputopt* option of the PLOP command-line tool to give the requested action priority over the input property. In the example above, the PDF/A conformance entry is removed from the document to allow encryption.

There are several combinations of input document properties and requested actions. In all of these combinations you can use the *sacrifice* option to allow an operation by sacrificing a particular document property (see Table 8.4, for details):

- PDF/A: PLOP applies digital signatures in a PDF/A-conforming manner: input documents which conform to the PDF/A-1, PDF/A-2 or PDF/A-3 standard are guaranteed to produce PDF/A-conforming signed output. However, encryption is not allowed for PDF/A documents since the standard prohibits any encryption. You can sacrifice PDF/A conformance with the *sacrifice={pdfa}* option, though. PDF pages used for signature visualization must also conform to PDF/A (see Section 7.3.1, »Visualizing Signatures with a Graphic or Logo«, page 102).
- PDF/X: PDF/X-1a/3/4/5 don't allow encryption or visible signature fields on the page. In these situations PLOP raises an exception, but you can sacrifice PDF/X conformance with the *sacrifice={pdfx}* option. Signature visualization is not supported in PDF/X mode.
- PDF/UA: most PLOP operations conform to PDF/UA-1 with the exception of *permissions=noaccessible*. You can sacrifice PDF/UA conformance with the *sacrifice={pdfua}* option.
- PLOP DS cannot apply signatures if the document contains non-signature form fields without appearances (e.g. form fields created with PDFlib 9.2 or earlier), and therefore issues an error for this kind of input. The reason is that Acrobat writes the missing form field representation into the document which instantly invalidates the signature. You can sacrifice existing form fields in this situation with the options *sacrifice={fields} update=false* in *create_document( )* or the *--outputopt* option of the PLOP command-line tool. The form field restriction does not apply to a signature field which will hold the generated signature. There are no restrictions for form field documents created with PDFlib 9.3 or above.
- If an unencrypted document contains encrypted file attachments for which the password is not available, processing stops by default. You can sacrifice encrypted file attachments with the option *sacrifice={encryptedattachments}* in *create_document( )* or the *--outputopt* option of the PLOP command-line tool. Encrypted file attachments for which the password is not available are removed with this option.
- If the input document contains one or more digital signatures and no new signature is created in update mode, processing stops with an exception by default. You can sacrifice existing signatures with the option *sacrifice={signatures}* in *create_document( )* or the *--outputopt* option of the PLOP command-line tool.

Properties of the input document which are generally lost. The following properties of the input document are lost after applying any PLOP operation:

- If the input document is linearized, the linearization is lost by default. In order to linearize the output, supply the *linearize* option to *create_document( )* or the *--linearize* option to the PLOP command-line tool. Note that linearization cannot be combined with digital signatures.
- Reader-enabled documents: processing Reader-enabled PDF documents with PLOP results in output which is not Reader-enabled. Since Reader-enabled documents can only be created with Adobe software there is no workaround.

Large PDF Documents. Although most users won't see any need for PDF documents in the range of Gigabytes, some enterprise applications must create or process documents containing a large number of, say, invoices or statements. While PLOP itself does not impose any limits on the size of the generated documents, there are several restrictions mandated by the PDF Reference and some PDF standards:

- 2 GB file size limit: PDF/A and other standards limit the file size to 2 GB. If a document gets larger than this limit, PLOP throws an exception when creating PDF/A, PDF/X-4 or PDF/X-5 output. Otherwise documents beyond 2 GB can be created.
- 10 GB file size limit: classical cross-reference tables in PDF documents are limited to 10 decimal digits and therefore 10^{10}-1 bytes, which equates to roughly 9.3 GB. However, this limit can be exceeded with compressed object streams. While compressed object streams reduce the overall file size anyway, the compressed cross-reference streams which are part of the *objectstreams* implementation are no longer subject to the 10-decimal-digits limit and therefore allow creation of PDF documents beyond 10 GB.
- Number of objects: while the object count in a document is not limited by PDF in general, the PDF/A, PDF/X-4 and PDF/X-5 standards limit the number of indirect objects in a document to 8.388.607. If a document requires objects beyond this limit, PLOP throws an exception when creating PDF/A, PDF/X-4 or PDF/X-5 output. In other modes documents with more objects can always be created. This check can be disabled with the option *limitcheck=false*.

What you can't do with PLOP. Please be aware of the following restrictions:
- PLOP is not a cracker tool – it cannot be used to gain access to protected documents without knowing the appropriate master password.
- You cannot process dynamic XFA forms – also called Adobe Experience Manager (AEM) forms – since these are not genuine PDF documents but rather XML forms packaged inside a thin PDF layer.

2 PLOP DS Features (Digital Signature)

Note The ability to digitally sign PDF documents is only available in PLOP DS, but not in the PLOP base product.

Digital signatures for PDF documents are covered in detail in Chapter 7, »Digital Signatures with PLOP DS«, page 87. In the current chapter we provide a summary and initial examples which may serve as a starting point.

2.1 Signature Features in PLOP DS

PDF Signature Properties.

- Create signatures in existing PDF signature fields or generate new fields which hold the signature. The signatures can be invisible or visible at a particular location on the page.
- Visualize digital signatures by importing a logo, scan of a handwritten signature or other representation as PDF page.
- Create PDF certification (author) signatures which allow document changes such as form-filling without breaking the signature.
- Validation information can be stored directly in the signature according to ISO 32000-1 or in a Document Security Store (DSS) as specified in ISO 32000-2 and PAdES part 4.
- Signatures can be applied in an incremental PDF update section to preserve existing signatures and document structure, or by rewriting the document structure which allows optimization and encryption.

PDF Versions and Standards. PLOP DS supports all relevant PDF versions and standards:

- PLOP DS processes all PDF versions up to Acrobat DC, i.e. PDF 1.7 (ISO 32000-1) up to extension level 8. PLOP DS can also process documents according to PDF 2.0 (ISO 32000-2).
- PLOP DS is aware of the PDF/A-1/2/3 (ISO 19005) archiving standards: if the input document conforms to PDF/A, the output document is guaranteed to conform as well. PLOP DS fully supports XMP extension schemas as required by PDF/A. The ability to insert PDF/A-conforming XMP metadata in PDF documents is an important advantage of PLOP DS.
- Similarly, PLOP DS is aware of the PDF/X-1a/3/4/5 (ISO 15930) print production standards, PDF/VT-1 (ISO 16612-2) for transactional printing and PDF/UA-1 (ISO 14289) for accessible PDF.

Signature standards.

- CMS-based PDF signatures according to PDF 1.7 (ISO 32000-1)
- Signatures for Long-Term Validation (LTV) according to PDF 2.0 (ISO 32000-2)
- PAdES (PDF Advanced Electronic Signatures) according to ETSI TS 102 778 part 2, 3 and 4, ETSI EN 319 142 and CAdES (ETSI TS 101 733) for qualified eIDAS signatures

PAdES conformance levels.

- Basic Signature (PAdES Level B-B)
- Signature with Time (PAdES Level B-T)
- Signature with Long-Term Validation Material (PAdES Level B-LT)
- Signature providing Long Term Availability and Integrity of Validation Material (PAdES Level B-LTA): required for eIDAS conformance
- Basic Electronic Signature (PAdES Level E-BES) and Explicit Policy-based Electronic Signature (PAdES Level E-EPES) according to PAdES part 3

Timestamping.

- Retrieve a timestamp from a trusted Timestamp Authority (TSA) according to RFC 3161, RFC 5816 and ETSI EN 319 422, and embed it in the generated signature. TSA details can be read from AATL certificates to create timestamps without any configuration.
- Create document-level timestamp signatures according to ISO 32000-2 and PAdES part 4. A document-level timestamp assures the state of a document without applying a personal signature.

Cryptographic Signature Details.

- Signatures according to the RSA and DSA algorithms as well as the Elliptic Curve Digital Signature Algorithm (ECDSA). PKCS#1 v1.5 and PKCS#1 v2.1 (PSS) encoding for RSA are supported.
- Strong signature and hash functions.
- Embed the full certificate chain in the generated signatures, which means that signatures with certificates from a CA (Certification Authority) on the Adobe Approved Trust List (AATL) or the European Union Trust List (EUTL) can be validated in Acrobat and Adobe Reader without any configuration on the client side.
- Embed Online Certificate Status Protocol responses (OCSP according to RFC 2560 and RFC 6960) and Certificate Revocation Lists (CRL according to RFC 3280) as revocation information for Long-Term Validation (LTV).

Signature Engines. PLOP DS supports multiple cryptographic engines, i.e. components for generating digital signatures:

- The built-in engine implements the required cryptographic functions directly in PLOP DS without any external dependencies. The built-in engine supports software-based digital IDs in the PKCS#12 and PFX formats.
- PLOP DS can attach cryptographic tokens via the standard PKCS#11 interface. This way digital IDs on smartcards, USB sticks, and other secure devices can be used for signing. This includes devices with an integrated keyboard for secure PIN input.
- The PKCS#11 interface can also be used to sign with a Hardware Security Module (HSM). HSMs offer secure key storage and ample performance for high-volume signing applications. PLOP DS uses PKCS#11 sessions to maximize performance of bulk signatures with HSMs. PLOP DS can also be used with HSMs in the cloud such as AWS CloudHSM.
- On Windows PLOP DS can leverage the cryptographic infrastructure provided by the operating system (MS CAPI). Digital IDs from the Windows certificate store can be used for signing, including software-based digital IDs and secure hardware tokens. Note that not all signature features are available with the MSCAPI engine, e.g. LTV.

- Alternatively a user-supplied cryptographic engine can be used to ensure that all cryptographic operations (hashing and signing) are performed in a dedicated cryptographic library.

What you can't do with PLOP DS. Please be aware of the following restrictions:
- You cannot Reader-enable PDF documents (e.g. allow annotation creation in Adobe Reader) with PLOP DS because this requires a specific Adobe signature.
- You cannot sign static or dynamic XFA forms.

2.2 Preparations for PLOP DS Evaluation

Install PDFlib Demo CA certificate in Acrobat. The following step is not required for creating digital signatures with PLOP DS. However, if you are evaluating PLOP DS with the sample certificates provided in the packages it is recommended to configure Acrobat as detailed below. This is not required if you work with certificates from a commercial CA which is installed in Acrobat's list of trusted certificates (see »Trusted Root Certificates in Acrobat«, page 90)

The sample certificates which are included in the PLOP DS package have been issued and signed by the PDFlib Demo CA. If you make the self-signed root certificate of this CA available to Acrobat, the generated signatures are accepted as fully valid in Acrobat. Proceed as follows for installing the PDFlib Demo CA certificate in Acrobat DC:
- Click *Edit, Preferences, Signatures, Identities & Trusted Certificates, More..., Trusted Certificates, Import, Browse...*
- Browse to *bind/data/PDFlibDemoCA_G3.crt* (part of the PLOP installation) and click *Import, Ok*.
- Now the entry *PDFlib GmbH Demo CA G3* is visible in the list of trusted certificates. Select this entry, click on *Edit Trust*, and activate the buttons *Use this certificate as a trusted root* and *Certified documents*, and click *Ok*.

Import demo digital IDs in Windows. In order to test the MSCAPI-based signature engine of PLOP DS on Windows you must make available digital IDs in the Windows certificate store. In order to import the demo digital IDs double-click on the corresponding *.p12* file to launch the certificate import wizard, and follow its instructions.

2.3 Signing Documents with PLOP DS

Applying a signature requires a digital ID, which may be available as a file, in the Windows certificate store, or on a cryptographic token (e.g. a smartcard or USB stick). While the former requires a password for accessing the digital ID, the Windows certificate store is usually protected by the Windows login and does not require any password. Cryptographic tokens are often protected by a PIN which must be supplied either by the signing software or directly on the token's integrated keyboard.

You can prepare a digital signature with *prepare_signature( )* which supports several options, and then apply it with *create_document( )*. Sample code for signing PDF documents is available in the *sign* and *multisign* mini samples which are included in all PLOP packages. The equivalent option for the PLOP command-line tool is *--signopt*.

Basic signature option list examples. Create an invisible signature for a PDF document using a digital ID from the file *demo_signer_rsa_2048.p12*. The password *demo* for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt" ↵
       --outfile signed.pdf input.pdf
```

(Windows only) Create an invisible signature for a PDF document using a certificate from the Windows Certificate Store (from the default store *My)*. This assumes that the digital ID is protected by your Windows login so that no password must be supplied:

```
plop --signopt "engine=mscapi digitalid={store=My subject={PLOP Demo Signer RSA-2048}}" ↵
        --outfile signed.pdf input.pdf
```

Create an invisible signature for a PDF document using a digital ID from a cryptographic token. The PKCS#11 interface for the token is implemented in the library *cryptoki.dll* which must be provided by the smartcard supplier. The password for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "engine=pkcs#11 digitalid={filename=cryptoki.dll} passwordfile=pw.txt" ↵
       --outfile signed.pdf input.pdf
```

More details are available in Section 7.2, »Signing with PLOP DS«, page 93.

2.4 Certification Signatures

A certification or author signature certifies the state of the document as the author created it while at the same time allowing certain changes without breaking the certification. The *certification* option specifies the changes which can be applied to the certified document without breaking the signature, e.g. form-filling allowed:

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
           certification=formfilling" ↵
           --outfile certified.pdf input.pdf
```

2.5 Timestamps

In order to add a timestamp to a signature you need the URL of a Timestamp Authority (TSA) and must supply it to the *timestamp* option:

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
       timestamp={source={url={http://timestamp.acme.com/tsa-noauth/tsa}}}" ↵
       --outfile signed.pdf input.pdf
```

Similarly, a document-level timestamp can be applied with the *doctimestamp* option:

```
plop --signopt ↵
       "doctimestamp={source={url={http://timestamp.acme.com/tsa-noauth/tsa}}}" ↵
       --outfile signed.pdf input.pdf
```

More details are available in Section 7.5, »Timestamps«, page 118.

2.6 LTV-enabled Signatures

Support for long-term validation (LTV) requires that all certificates in the chain are available, and that certificate revocation information can be obtained online or from a disk file when the signature is created. This requires suitable OCSP or CRL servers to be provided by the PKI. In many cases (especially AATL certificates) the necessary network information can be read from the signing certificate. Otherwise you must provide suitable network resources via the *ocsp* and/or *crl/crlfile/crldir* options. In order to provide access to the whole certificate chain you must supply the option *rootcertfile* with the name of a PEM file containing the root CA certificate.

LTV-enabled signatures generally require online PKI resources (CRL or OCSP) for the certificates in use, which are not available for the PLOP DS demo certificates. As a workaround you can use the CRL file *PDFlibDemoCA_G3.crl* which is provided in the distribution (this CRL has a very long expiration date which would not be acceptable in a production environment). The corresponding command-line call for creating LTV-enabled signatures looks as follows:

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} password=demo ltv=full ↵
        crlfile=PDFlibDemoCA_G3.crl rootcertfile=PDFlibDemoCA_G3.pem" ↵
        --outfile ltv-signed.pdf input.pdf
```

For the next example we assume that the required OCSP or CRL retrieval information is present in the signing certificate, which is typically the case for commercial certificates. Under these conditions you can supply the option *ltv=full* to make sure that an LTV-enabled signature is created:

```
plop --signopt "digitalid={filename=signer.p12} passwordfile=pw.txt ltv=full ↵
        rootcertfile=RootCA.pem" --outfile ltv-signed.pdf input.pdf
```

Note that this may not be sufficient depending on the details of the involved PKI. In particular, revocation information must also be available for the OCSP/CRL signer and the timestamp authority.

2.7 PAdES Signatures

The PAdES family of signature standards improves PDF signatures and ensures that EU requirements are met. Various signature options can be used to create signatures according to different PAdES flavors. For example, the following command-line creates a basic signature according to PAdES part 3 (PAdES Level B-B):

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
        --outfile signed.pdf input.pdf
```

The following command-line creates a signature according to PAdES part 3 with explicit policy identifier (PAdES Level E-EPES):

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
        policy={oid=2.16.276.1.89.1.1.1.1.3 commitmenttype=origin}" ↵
        --outfile signed.pdf input.pdf
```

More details are available in Section 7.7, »The CAdES and PAdES Signature Standards«, page 129.

2.8 Visualize Digital Signatures

A digital signature can be visualized, e.g. by a company logo or the scan of a handwritten signature. The visual representation must be supplied as a PDF document which will be placed in the signature form field. If the input document does not yet contain a signature field suitable field coordinates must be supplied. The following command-line places the visualization document *signing_man.pdf* in the field rectangle:

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
        field={name=Signature1 rect={10 10 adapt adapt}}" --visdoc signing_man.pdf ↵
        --outfile signed.pdf input.pdf
```

More details are available in Section 7.3.1, »Visualizing Signatures with a Graphic or Logo«, page 102.

2.9 Query Digital Signatures

Querying general signature properties. With the pCOS programming interface which is integrated in PLOP DS you can query signature settings of a PDF document. The pCOS Cookbook topic *signatures* demonstrates how to query signature types and details. Sample code for querying document information with pCOS can be seen in the *dumper* mini sample, which is included in all PLOP packages. The pCOS command-line tool can be used to query information from PDF documents without any programming (see Section 1.6, »Query Document Information with pCOS«, page 22):

```
pcos *.pdf
```

This program call creates output similar to the following:

```
       File name: hellosign.pdf
       File size: 166699
     PDF version: 1.7
       Revisions: 0
       Master pw: false
         User pw: false

      ...
      Tagged PDF: false
      Signatures: 1
        signature field 'Signature1': invisible approval signature, CAdES
  Reader-enabled: false
```

When working with the pCOS programming interface you can use the *signaturefields[ ]* pseudo object to retrieve details about the signatures in a PDF document (see pCOS Path Reference for details).

Extracting CMS signature objects. The following command extracts the CMS object with cryptographic details in DER format from a signed document:

```
pcos --binary --pcospath signaturefields[0]/V/Contents[0]/V/Contents ↵
        --outfile signature.der input.pdf
```

The extracted DER-encoded CMS object can further be analyzed, e.g. with the OpenSSL command-line tool:

```
openssl cms -in signature.der -inform DER -cmsout -print
```

3 PLOP and PLOP DS Command-line Tool

Note Also take a look at the pCOS command-line tool which is discussed in a separate manual.

3.1 PLOP and PLOP DS Command-line Options

The combined command-line tool for PLOP and PLOP DS allows you to encrypt, decrypt, optimize, repair, and sign one or more PDF documents without the need for any programming. In addition, it can be used to query the status of PDF documents. The PLOP program can be controlled via a number of command-line options. It is called as follows for one or more input PDF files (items in square brackets are optional):

```
plop --help
plop [ <general options> ] <transform options> --outfile <filename> <filename>
plop [ <general options> ] <transform options> --targetdir <pathname> <filename>...
```

The PLOP command-line tool is built on top of the PLOP library. By default PLOP repairs input documents which are found to be damaged. You can supply library options using the *--inputopt, --outputopt, --plopopt,--signopt* and *--visdocopt* options according to the option tables in Chapter 8, »PLOP and PLOP DS Library API Reference«, page 135. Table 3.1 lists all PLOP command-line options.

Table 3.1 PLOP command-line options

option	*parameters*	*function*
--		*End the list of options; useful for file names which start with a - character.*
@filename[1]		*Specify a response file with the name* `filename` *which contains options; for a syntax description see »Response files«, page 41. Response files will only be recognized before the* `--` *option and before the first filename, and can not be used to replace the parameter for another option.*
--help, -? ***(or no option)***		*Display help with a summary of available options.*
--inputopt	<*option list*>	*Option list for open_document( ) (see Table 8.3, page 141)*
--master, -m	<*password*>	*Output master password; missing option means no password*
--noreplace, -n		*If the output file already exists, it will not be overwritten and an exception will be thrown. Default: existing output files will be overwritten.*
--outfile, -o	<*filename*>	*(Requires exactly one input document; one of* `--outfile` *and* `--targetdir` *must be supplied) Output file name; input and output file name must be different.*
--outputopt	<*option list*>	*Option list for create_document( ) (see Table 8.4, page 146)*
--password, -p	<*password*>	*User or master password for input document(s). This password is used for all input documents. Documents which require different passwords must be processed in separate program calls.* *If a digital ID has been supplied with* `--inputopt`*, the password is applied to the ID.*

Table 3.1 PLOP command-line options

option	*parameters*	*function*
--permissions	<*permissions*>	*(Requires* `--master or --recipient`*) Access permission list for the output document. It contains any number of the* `noprint`, `nomodify`, `nocopy`, `noannots`, `noassemble`, `noforms`, `noaccessible`, `nohiresprint`, *and* `plainmetadata` *keywords (see Table 5.3, page 66). In addition, the following keyword can be used (default: no permission restrictions):* ***keep*** *Keep the permission settings of the input document. This setting can be amended by additional keywords in order to modify the permission settings of the input PDF, e.g.* `keep noprint`. *In certificate security mode (option* `--recipient`*) only* `plainmetadata` *is allowed. Other permission restrictions can be specified in the* `permissions` *option of the* `--recipient` *option list.*
--plopopt	<*option list*>	*Option list for set_option( ) (see Table 8.12, page 165). This can be used to pass the* `license` *or* `licensefile` *options.*
--recipient, -r[1]	<*option list*>	*Option list for add_recipient( ) to add a recipient for documents protected with certificate security. Supplying this option at least once activates certificate security mode. The recipient is used for all input files. This option must not be combined with* `--master/-m` *and* `--user/-u`.
--recsize	<*blocksize*>	*(MVS only) Record size of the output file. Default: 0 (unblocked)*
--searchpath, -s[1]	<*path*>	*Name of a directory where files will be searched. The path must not start with a minus character »-« (prepend* `.` */ if required). Default: current directory*
--signopt, -S	<*option list*>	*(Only available in PLOP DS) Option list for prepare_signature( ) for digitally signing documents (see Table 8.7, page 153).*
--targetdir, -t	<*dirname*>	*(One of* `--outfile` *and* `--targetdir` *must be supplied) Output directory name; the directory must already exist.*
--tempdirname	<*dirname*>	*Name of a directory where temporary files needed for PLOP's internal processing will be created. If empty, PLOP will generate temporary files in the current directory. Default: empty*
--tempfilename, -T	<*filename*>	*(MVS only) Full file name for a temporary file for PLOP's internal processing. If empty, PLOP will generate a unique temporary file name. The user is responsible for deleting the temporary file when PLOP finished. Default: empty*
--user, -u	<*password*>	*(Requires* `--master`*) Output user password; missing option means no password*
--verbose, -v	0, 1, 2, 3	*Verbosity level (default: 1):* ***0*** *no output* ***1*** *only error messages* ***2*** *add file names and API method name in error messages* ***3*** *detailed reporting*
--visdoc	<*filename*>	*(Only with* `--signopt`*) Name of the PDF file from which a page will be used for visualizing the digital signature.*
--visdocopt	<*option list*>	*(Only with* `--visdoc`*) Option list for open_document( ) (see Table 8.3, page 141) which is used to open the signature visualization document*
--webopt, -w		*Linearize the PDF output for Web delivery, also known as Web optimization. Default: no linearization*

1. This option can be supplied more than once.

Constructing PLOP command lines. The following rules must be obeyed for constructing PLOP command lines:

- Input files will be searched in all directories specified as *searchpath*.
- Short forms are available for some options, and can be mixed with long options.
- Long options can be abbreviated provided the abbreviation is unique (e.g. *--plop* instead of *--plopopt)*.
- If an option is supplied more than once only the last instance will be taken into account. However, this rule does not hold for options which are marked as repeatable in Table 3.1.
- Depending on the encryption status of the input file, a user or master password may be required for processing. This must be supplied with the *--password* option. PLOP will check whether this password is sufficient for the requested action (see Table 5.2), and will throw an exception if it isn't.

PLOP checks the full command line before processing any file. If an option syntax error is encountered in the options anywhere on the command line, no files will be processed at all. If a particular file cannot be processed (e.g. because the required password is missing), an error message will be emitted, and PLOP will continue processing the remaining files.

File names. File names which contain blank characters require some special handling when used with command-line tools like PLOP. In order to process a file name with blank characters you should enclose the complete file name with double quote " characters. Wildcards can be used according to standard practice. For example, **.pdf* denotes all files in a given directory which have a *.pdf* file name suffix. Note that on some systems case is significant, while on others it isn't (i.e., **.pdf* may be different from **.PDF)*. Also note that on Windows systems wildcards do not work for file names containing blank characters. Wildcards will be evaluated in the current directory, not any searchpath directory.

On Windows all file name options accept Unicode strings, e.g. as a result of dragging files from the Explorer to a command prompt window.

Response files. In addition to options supplied directly on the command-line, options can also be supplied in a response file. The contents of a response file are inserted in the command-line at the location where the *@filename* option was found.

A response file is a simple text file with options and parameters. It must adhere to the following syntax rules:

- Option values must be separated with whitespace, i.e. space, linefeed, return, or tab.
- Values which contain whitespace must be enclosed with double quotation marks: "
- Double quotation marks at the beginning and end of a value are omitted.
- A double quotation mark must be masked with a backslash to use it literally: \"
- A backslash character must be masked with another backslash to use it literally: \\

Response files can be nested, i.e. *@filename* can be used in another response file.

Response files may contain Unicode strings for file name and password options. Response files can be encoded in UTF-8, EBCDIC-UTF-8, or UTF-16 format and must start with the corresponding BOM. If no BOM is found, the contents of the response file are interpreted in EBCDIC on IBM Z, and in ISO 8859-1 (Latin-1) on all other systems.

Exit codes. The PLOP command-line tool returns with an exit code which can be used to check whether or not the requested operations could be successfully carried out:

- Exit code 0: all command-line options and input files could be successfully and fully processed.
- Exit code 1: one or more file processing errors occurred, but processing continued.
- Exit code 2: some error was found in the command-line options. Processing stopped at the particular bad option, and no documents have been processed.

3.2 PLOP and PLOP DS Command-line Examples

The following examples demonstrate some useful combinations of PLOP command-line options. All samples are shown in two variations; the first uses the long format of all options, while the second uses the equivalent short option format. More examples are available in the following sections:

- Chapter 1, »PLOP Features«, page 17 (various sections);
- Section 5.3, »Applying Password Security on the Command-Line«, page 68;
- Section 6.5, »Applying Certificate Security on the Command-Line«, page 84;
- Section 7.2, »Signing with PLOP DS«, page 93.

Linearize all PDF documents in a directory (assuming these do not require any password), and copy the resulting files to the target directory *output*. Verbosity level 2 prints the names of all input and output files as they are processed:

```
plop --verbose 2 --webopt --targetdir output *.pdf
plop -v 2 -w -t output *.pdf
```

Encrypt all files in the current directory with the same user password *demo* and master password *DEMO*, and place the resulting files in the target directory *output:*

```
plop --targetdir output --user demo --master DEMO *.pdf
plop -t output -u demo -m DEMO *.pdf
```

Encrypt a document against a single recipient certificate:

```
plop --recipient "certificate={filename=demo_recipient_1.pem}" ↵
       --outfile protected.pdf input.pdf
plop -r "certificate={filename=demo_recipient_1.pem}" -o protected.pdf input.pdf
```

Create an invisible signature for a PDF document, using a digital ID from the file *demo_signer_rsa_2048.p12*. The password for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt" ↵
       --outfile signed.pdf input.pdf
plop -S "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt" ↵
       -o signed.pdf input.pdf
```

Create a signature, using an existing PDF with a hand-written signature to visualize the signature:

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
       field={rect={100 100 300 adapt}}" --visdoc signature.pdf ↵
       --outfile signed.pdf input.pdf
plop -S "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
       field={rect={100 100 300 adapt}}" --visdoc signature.pdf -o signed.pdf input.pdf
```

4 PLOP and PLOP DS Library Language Bindings

4.1 C Binding

PLOP is written in C with some C++ modules. In order to use the C binding you can use a static or shared library (DLL/SO), and you need the central PLOP include file *ploplib.h* for inclusion in your client source modules. Alternatively, *ploplibdl.h* can be used for dynamically loading the PLOP DLL at runtime (see next section for details).

Note *Applications which use the PLOP binding for C must be linked with a C++ compiler since the library includes some parts which are implemented in C++. Using a C linker may result in unresolved externals unless the application is explicitly linked against the required C++ support libraries.*

Error handling. The PLOP API provides a mechanism for acting upon exceptions thrown by the library in order to compensate for the lack of native exception handling in the C language. Using the *PLOP_TRY( )* and *PLOP_CATCH( )* macros client code can be set up such that a dedicated piece of code is invoked for error handling and cleanup when an exception occurs. These macros set up two code sections: the try clause with code which may throw an exception, and the catch clause with code which acts upon an exception. If any of the API methods called in the try block throws an exception, program execution will continue at the first statement of the catch block immediately. The following rules must be obeyed in PLOP client code:

- *PLOP_TRY( )* and *PLOP_CATCH( )* must always be paired.
- *PLOP_new( )* will never throw an exception; since a try block can only be started with a valid PLOP object handle, *PLOP_new( )* must be called outside of any try block.
- *PLOP_delete( )* will never throw an exception, and therefore can safely be called outside of any try block. It can also be called in a catch clause.
- Special care must be taken about variables that are used in both the try and catch blocks. Since the compiler doesn't know about the transfer of control from one block to the other, it might produce inappropriate code (e.g., register variable optimizations) in this situation.
  Fortunately, there is a simple rule to avoid this kind of problem: Variables used in both the try and catch blocks must be declared *volatile*. Using the *volatile* keyword signals to the compiler that it must not apply dangerous optimizations to the variable.
- If a try block is left (e.g., with a return statement, thus bypassing the invocation of the corresponding *PLOP_CATCH( )*), the *PLOP_EXIT_TRY( )* macro must be called before the return statement to inform the exception machinery.
- As in all PLOP language bindings document processing must stop when an exception was thrown.

The following code fragment demonstrates these rules with the typical idiom for dealing with PLOP exceptions in client code (full samples can be found in the PLOP package):

```
if ((plop = PLOP_new()) == (PLOP *) 0)
{
```

```
        printf("Couldn't create PLOP object out of memory\n");
        return(2);
}
PLOP_TRY(plop)
{
        /* statements that directly or indirectly call API methods */
}
PLOP_CATCH(plop)
{
        printf("Error %d in %s(): %s\n",
            PLOP_get_errnum(plop), PLOP_get_apiname(plop),PLOP_get_errmsg(plop));
}
PLOP_delete(plop);
```

Unicode handling for name strings. The C programming language supports genuine Unicode strings only in version C11. Since this version is not yet generally supported, PLOP/PLOP DS offers Unicode support based on the traditional *char* data type. Some string parameters for API methods may be declared as *name strings*. These are handled depending on the *length* parameter and the existence of a BOM at the beginning of the string. In C, if the *length* parameter is different from 0 the string will be interpreted as UTF-16. If the *length* parameter is 0 the string will be interpreted as UTF-8 if it starts with a UTF-8 BOM, or as EBCDIC UTF-8 if it starts with an EBCDIC UTF-8 BOM, or as *host* encoding if no BOM is found (or *ebcdic* on EBCDIC-based platforms).

Unicode handling for option lists. Strings within option lists require special attention since they cannot be expressed as Unicode strings in UTF-16 format, but only as byte arrays. For this reason UTF-8 is used for Unicode options. By looking for a BOM at the beginning of an option PLOP decides how to interpret it. The BOM will be used to determine the format of the string. More precisely, interpreting a string option works as follows:

- If the option starts with a UTF-8 BOM *(\xEF\xBB\xBF)* it will interpreted as UTF-8.
- If the option starts with an EBCDIC UTF-8 BOM *(\x57\x8B\xAB)* it will be interpreted as EBCDIC UTF-8.
- If no BOM is found, the string will be treated as *winansi* (or *ebcdic* on EBCDIC-based platforms).

Note The PLOP_convert_to_unicode( ) utility method can be used to create UTF-8 strings from UTF-16 strings, which is useful for creating option lists with Unicode values.

Using PLOP as a DLL loaded at runtime. While most clients will use PLOP as a statically bound library or a dynamic library which is bound at link time, you can also load the DLL at runtime and dynamically fetch pointers to all API methods. This is especially useful to load the DLL only on demand. PLOP supports a special mechanism to facilitate this dynamic usage. It works according to the following rules:

- Include *ploplibdl.h* instead of *ploplib.h*.
- Use *PLOP_new_dl( )* and *PLOP_delete_dl( )* instead of *PLOP_new( )* and *PLOP_delete( )*.
- Use *PLOP_TRY_DL( )* and *PLOP_CATCH_DL( )* instead of *PLOP_TRY( )* and *PLOP_CATCH( )*.
- Use function pointers for all other PLOP calls.
- Compile the auxiliary module *ploplibdl.c* and link your application against the resulting object file.

The dynamic loading mechanism is demonstrated in the *encryptdl.c* sample.

4.2 C++ Binding

In addition to the *ploplib.h* C header file, an object-oriented wrapper for C++ is supplied for PLOP clients. It requires the *plop.hpp* header file, which in turn includes *ploplib.h*. Since *plop.hpp* contains a template-based implementation no corresponding *plop.cpp* module is required. Using the C++ object wrapper replaces the functional approach with API methods and *PLOP_* prefixes in all PLOP function names with a more object-oriented approach.

String handling in C++. PLOP's template-based approach supports the following usage patterns with respect to string handling:

- Strings of the C++ standard library type *std::wstring* are used as basic string type. They can hold Unicode characters encoded as UTF-16 or UTF-32. This is the default behavior and the recommended approach for new applications unless custom data types (see next item) offer a significant advantage over *wstrings*.
- Custom (user-defined) data types for string handling can be used as long as the custom data type is an instantiation of the *basic_string* class template and can be converted to and from Unicode via user-supplied converter methods.

The default interface assumes that all strings passed to and received from PLOP methods are native *wstrings*. Depending on the size of the *wchar_t* data type, *wstrings* are assumed to contain Unicode strings encoded as UTF-16 (2-byte characters) or UTF-32 (4-byte characters). Literal strings in the source code must be prefixed with *L* to designate wide strings. Unicode characters in literals can be created with the *\u* and *\U* syntax. Although this syntax is part of standard ISO C++, some compilers don't support it. In this case literal Unicode characters must be created with hex characters.

Note *On EBCDIC-based systems the formatting of option list strings for the* `wstring`*-based interface requires additional conversions to avoid a mixture of EBCDIC and UTF-16 wstrings in option lists. Convenience code for this conversion and instructions are available in the auxiliary module* `utf16num_ebcdic.hpp`.

Error handling in C++. PLOP API methods throw a C++ exception in case of an error. These exceptions must be caught in the client code by using C++ *try/catch* clauses. In order to provide extended error information the PLOP class provides a public *PLOP::Exception* class which exposes methods for retrieving the detailed error message, the exception number, and the name of the PLOP API method which threw the exception.

Native C++ exceptions thrown by PLOP routines will behave as expected. The following code fragment will catch exceptions thrown by PLOP:

```
try {
        ...PLOP instructions...
} catch (PLOP::Exception &ex) {
            wcerr << L"Error " << ex.get_errnum()
            << L" in " << ex.get_apiname()
            << L"(): " << ex.get_errmsg() << endl;
}
```

Using PLOP as a DLL loaded at runtime. Similar to the C language binding the C++ binding allows you to dynamically attach PLOP to your application at runtime (see »Us-

ing PLOP as a DLL loaded at runtime«, page 46). Dynamic loading can be enabled as follows when compiling the application module which includes *plop.hpp:*

```
#define PLOPCPP_DL    1
```

In addition you must compile the auxiliary module *ploplibdl.c* and link your application against the resulting object file. Since the details of dynamic loading are hidden in the PLOP object it does not affect the C++ API: all method calls look the same regardless of whether or not dynamic loading is enabled.

4.3 Java Binding

Installing the PLOP Edition for Java. PLOP/PLOP DS has been implemented as a native C library which attaches to Java via the JNI (Java Native Interface). Obviously, for developing Java applications you will need the JDK which includes support for the JNI. For the PLOP binding to work, the Java VM must have access to the PLOP Java wrapper library and the PLOP Java package.

The PLOP Java package. In order to maintain a consistent look-and-feel for the Java developer, PLOP is organized as a Java package with the following package name:

```
com.pdflib.plop
```

This package is available in the *plop.jar* file and contains a single class called *plop*. Last-minute comments on using PLOP in various Java development environments may be found in the *readme.txt* file.

In order to supply this package to your application, you must add *plop.jar* to your *CLASSPATH* environment variable, add the option *-classpath plop.jar* in your calls to the Java compiler and runtime, or perform equivalent steps in your Java IDE. You can configure the Java VM to search for native libraries in a given directory by setting the *java.library.path* property to the name of the directory, e.g.

```
java -Djava.library.path=. encrypt
```

You can check the value of this property as follows:

```
System.out.println(System.getProperty("java.library.path"));
```

In addition, the following platform-dependent steps must be performed:

- Unix: The library *libplop_java.so* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.
- macOS: The library *libplop_java.jnilib* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.
- Windows: The library *plop_java.dll* must be placed in the Windows system directory, or a directory which is listed in the PATH environment variable.

Exception Handling in Java. All PLOP/PLOP DS methods will throw an exception of type *PLOPException* in case of an error. PLOP users can use standard Java language features to catch the exception and react on it:

```
try {
        plop plop;
        /* ... PLOP statements ... */

} catch (PLOPException e) {
        System.err.println("encrypt: PLOP Exception occurred:");
        System.err.println(e.get_apiname() +": " + e.get_errmsg());
} catch (Exception e) {
        System.err.println(e);
} finally {
        /* delete the PLOP object */
        if (plop != null) plop.delete();
}
```

4.4 .NET Binding

Note The .NET binding is delivered as a universal package for all supported platforms. However, it still requires a platform-specific license key which cannot be transferred across platforms.

Table 4.1 describes general aspects of the .NET binding.

Table 4.1 General properties of the .NET binding

Topic	***Implementation of the .NET binding***
.NET support	*.NET Standard 2.0 and above, e.g. .NET Core 2/3, .NET 5/6*
.NET Framework support	*.NET Framework 4.6.1 and above*
target platforms	*Windows, Linux, macOS; see* `system-requirements.txt` *for exact list of platform/architecture combinations*
download package	*universal* `zip` *package for all supported platforms*
package contents	*NuGet package with managed and native libraries, documentation, and samples*
implementation	*C# assembly* `PLOP_dotnet.dll` *with managed code and platform-specific shared library* `PLOP_dotnet_native.[dll|so|dylib]` *with native code*
.NET integration	*C# Interop via explicit PInvoke*
versioning scheme	*The versioning scheme conforms to .NET rules. The .NET version numbers are visible e.g. in the NuGet cache and* `.csproj` *project files. They differ from the product's major and minor release numbers. A mapping between both versioning schemes can be found in* `compatibility.txt`.

Installing PLOP.NET. The product is supplied as NuGet package where the package name includes the version number. Internal the packages for .NET use semantic version according to .NET rules. As a result, the internal version numbers differ from the regular product version number. A mapping table for both numbering schemes is available in *compatibility.txt*.

The NuGet package can be installed locally using any of the following methods:

- The *dotnet* command-line tool (all platforms). This method is detailed in the next section.
- Visual Studio's Package Manager UI (Windows and macOS)
- Visual Studio's Package Manager Console (Windows)
- The *nuget* command-line tool (all platforms)

The project files for the supplied samples are prepared for target framework .NET 6.0 (target framework moniker TFM=*net6.0)*.

Installing PLOP for .NET with the dotnet command-line tool. We describe the installation, configuration and build process with the *dotnet* utility, using the supplied *hello* project as an example:

- Unpack the product package in a directory of your choice.
- In a command shell *cd* to the *hello* project directory:

```
cd <installdir>\bind\dotnet\csharp\hello
```

- (This step is not required for the supplied samples which reference the package with a local *NuGet.Config* file) Copy the NuGet package to the application's project directory:

  ```
  <installdir>/bind/dotnet/PLOP_dotnet.X.Y.Z.nupkg
  ```

- (This step is not required for the supplied samples which already contain a reference to PLOP) Enter the following command with the appropriate version number (the version number can be found in the name of the *.nupkg* file):

  ```
  dotnet add package PLOP_dotnet.X.Y.Z
  ```

  This command adds a PLOP reference to the *.csproj* project file. It also installs PLOP in the local NuGet package cache if it is not yet present, e.g.

  ```
  ~/.nuget/packages/plop_dotnet/X.Y.Z
  ```

  Because of this caching you must copy the **.nupkg* only for the first project. Subsequent projects don't require the package file since it is taken from the cache.
- Now you can build and run the *hello* project to test it:

  ```
  dotnet build
  dotnet run
  ```

  As a result you will find the generated *hello.pdf* output document in the application directory.

Full examples with ready-to-use configuration are included in all packages. Once the .NET binding is properly referenced you can use the *PLOP_dotnet*.PLOP and *PLOP_dotnet.PLOPException* classes.

Installing PLOP.NET in a .NET Framework project. Proceed as follows to add PLOP.NET to a .NET Framework 4.x project with Visual Studio:

- Add a reference to *PLOP_dotnet.x.y.z.nupkg* with the NuGet Package manager. As a result a reference to *PLOP_dotnet* is added to your project as well as a *packages.config* management file.
- In the context menu of *packages.config* choose *Migrate packages.config to PackageReference...* to convert the reference (which initially points only to *PLOP_dotnet.dll)* to a NuGet package reference.
- In the *Configuration Manager* create a *New Solution Platform* with the name *x86* (32-bit) or *x64* (64-bit). Do not use the setting *Any CPU* since it won't work.
- If you plan to use the built-in IIS the selected Bitness must match the platform; set *x86* or *x64* appropriately via *Project, Properties, Web, Servers*.
- If you change the configuration you must rebuild the project to ensure that the correct version of *PLOP_dotnet_native.dll* is copied to the *bin* directory.

Error handling in .NET. The .NET binding supports .NET exceptions and will throw an exception with a detailed error message when a runtime problem occurs. The client is responsible for catching such an exception and properly reacting on it. Otherwise the .NET framework will catch the exception and usually terminate the application.

In order to convey exception-related information PLOP defines its own exception class *PLOP_dotnet.PLOPException* with the members *get_errnum, get_errmsg,* and *get_apiname*. PLOP implements the *IDisposable* interface which means that clients can call the *Dispose( )* method for cleanup.

Client code can handle .NET exceptions thrown by PDFlib with the usual *try...catch* construct:

```
try {
        p = new PLOP();
        ...PLOP instructions...
} catch (PLOPException e) {
        // caught exception thrown by PLOP
        Console.WriteLine("PLOP exception occurred:\n");
        Console.WriteLine("[{0}] {1}: {2}\n",
            e.get_errnum(), e.get_apiname(), e.get_errmsg());
} finally {
        if (p != null) {
            p.Dispose();
        }
}
```

4.5 Objective-C Binding

Although the C and C++ language bindings can be used with Objective-C, a genuine language binding for Objective-C is also available. The PLOP framework is available in the following flavors:

- *PLOP* for use on macOS
- *PLOP_ios* for use on iOS

Both frameworks contain language bindings for C, C++, and Objective-C.

Installing the PLOP Edition for Objective-C on macOS. In order to use PLOP in your application you must copy *PLOP.framework or PLOP_ios.framework* to the directory */Library/Frameworks*. Installing the PLOP framework in a different location is possible, but requires use of Apple's *install_name_tool* which is not described here. The *PLOP_objc.h* header file with PLOP method declarations must be imported in the application source code:

```
#import "PLOP/PLOP_objc.h"
```

or

```
#import "PLOP_ios/PLOP_objc.h"
```

In order to embed the PLOP/PLOP DS framework in an app, XCode's code signing expects a framework with the version number *A* while PDFlib products use numeric version numbers. In order to get around this you can create an appropriately named framework folder as follows:

```
cd PLOP.framework/Versions
mv 5.5 A
rm Current
ln -s A Current
```

Parameter naming conventions. For PLOP method calls you must supply parameters according to the following conventions:

- The value of the first parameter is provided directly after the method name, separated by a colon character.
- For each subsequent parameter the parameter's name and its value (again separated from each other by a colon character) must be provided. The parameter names can be found in Chapter 8, »PLOP and PLOP DS Library API Reference«, page 135, and in *PLOP_objc.h*.

For example, the following line in the API description:

```
int open_document(wstring filename, wstring optlist)
```

corresponds to the following Objective-C method:

```
- (NSInteger) open_document: (NSString *) filename optlist: (NSString *) optlist;
```

This means your application must make a call similar to the following:

```
doc = [plop open_document:filename optlist:pageoptlist];
```

Xcode Code Sense for code completion can be used with the PLOP framework.

Error handling in Objective-C. The Objective-C binding translates PLOP errors to native Objective-C exceptions. In case of a runtime problem PLOP throws a native Objective-C exception of the class *PLOPException*. These exceptions can be handled with the usual *try/catch* mechanism:

```
@try {
        ...PLOP instructions...
}
@catch (PLOPException *ex) {
        NSString * errorMessage =
            [NSString stringWithFormat:@"PLOP error %d in '%@': %@",
            [ex get_errnum], [ex get_apiname], [ex get_errmsg]];
        NSAlert *alert = [[NSAlert alloc] init];
        [alert setMessageText: errorMessage];
        [alert runModal];
        [alert release];
}
@catch (NSException *ex) {
        NSAlert *alert = [[NSAlert alloc] init];
        [alert setMessageText: [ex reason]];
        [alert runModal];
        [alert release];
}
@finally {
        [plop release];
}
```

In addition to the *get_errmsg* method you can also use the *reason* field of the exception object to retrieve the error message.

4.6 Perl Binding

The PLOP wrapper for Perl consists of a C wrapper and two Perl package modules, one for providing a Perl equivalent for each PLOP API method and another one for the PLOP object. The C module is used to build a shared library which the Perl interpreter loads at runtime, with some help from the package file. Perl scripts refer to the shared library module via a *use* statement.

Installing the PLOP Edition for Perl. The Perl extension mechanism loads shared libraries at runtime through the DynaLoader module. The Perl executable must have been compiled with support for shared libraries (this is true for the majority of Perl configurations).

For the PLOP binding to work, the Perl interpreter must access the PLOP Perl wrapper and the modules *plop_pl.pm* and *PDFlib/PLOP.pm*. In addition to the platform-specific methods described below you can add a directory to Perl's *@INC* module search path using the *-I* command line option:

```
perl -I/path/to/plop encrypt.pl
```

Unix. Perl will search *plop_pl.so* (on macOS: *plop_pl.bundle)*, *plop_pl.pm* and *PDFlib/PLOP.pm* in the current directory, or the directory printed by the following Perl command:

```
perl -e 'use Config; print $Config{sitearchexp};'
```

Perl will also search the subdirectory *auto/plop_pl*. Typical output of the above command looks like

```
/usr/lib/perl5/site_perl/5.32/i686-linux
```

Windows. The DLL *plop_pl.dll* and the modules *plop_pl.pm* and *PDFlib/PLOP.pm* will be searched in the current directory, or the directory printed by the following Perl command:

```
perl -e "use Config; print $Config{sitearchexp};"
```

Typical output of the above command looks like

```
C:\Program Files\Perl5.32\site\lib
```

Exception Handling in Perl. When a PLOP exception occurs, a Perl exception is thrown. It can be caught and acted upon using an *eval* sequence:

```
eval {
        ...PLOP instructions...
};
die "Exception caught: $@" if $@;
```

4.7 PHP Binding

Note *Detailed information about the various flavors and options for using PLOP with PHP can be found in the PDFlib-in-.NET-HowTo.pdf document which is contained in the distribution packages and also available on the PDFlib Web site. Although it is mainly targeted at using PDFlib with PHP the discussion applies equally to using PLOP with PHP.*

Installing the PLOP Edition for PHP. PLOP/PLOP DS is implemented as a C library which can dynamically be attached to PHP. PLOP supports several versions of PHP. Depending on the version of PHP you use you must choose the appropriate PLOP library from the unpacked PLOP archive.

You must configure PHP so that it knows about the PLOP library. Add the following line in *php.ini:*

```
extension=plop_php
```

PHP searches the library in the directory specified in the *extension_dir* variable in *php.ini* on Unix, and additionally in the standard system directories on Windows. You can test which version of the PHP PLOP binding you have installed with the following one-line PHP script:

```
<?phpinfo()?>
```

This will display a long info page about your current PHP configuration. On this page check the section titled *plop*. If this section contains the phrase

```
PDFlib PLOP (PDF Linearization, Optimization, Protection and Digital Signature) =>
enabled
```

(plus the PLOP version number) you successfully installed PLOP for PHP.

File name handling in PHP. Unqualified file names (without any path component) and relative file names for PDF, image, font and other disk files are handled differently in Unix and Windows versions of PHP:

- PHP on Unix systems will find files without any path component in the directory where the script is located.
- PHP on Windows will find files without any path component only in the directory where the PHP DLL is located.

Exception handling in PHP. Since PHP supports structured exception handling, PLOP exceptions will be propagated as PHP exceptions. You can use the standard *try/catch* technique to deal with PLOP exceptions:

```
try {

...PLOP instructions...

} catch (PLOPException $e) {
    print "PLOP exception occurred:\n";
    print "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": "
                $e->get_errmsg() . "\n";
}
catch (Throwable $e) {
```

```
    die("PHP exception occurred: " . $e->getMessage() . "\n");
}
```

Developing with Eclipse and Zend Studio. The PHP Development Tools (PDT) support PHP development with Eclipse and Zend Studio. PDT can be configured to support context-sensitive help with the steps outlined below.

Add PLOP to the Eclipse preferences so that it will be known to all PHP projects:

- Select *Window, Preferences, PHP, PHP Libraries, New...* to launch a wizard.
- In *User library name* enter *PLOP*, click *Add External folder...* and select the folder *bind\php\Eclipse PDT*.

In an existing or new PHP project you can add a reference to the PLOP library as follows:

- In the PHP Explorer right-click on the PHP project and select *Include Path, Configure Include Path...*
- Go to the *Libraries* tab, click *Add Library...*, and select *User Library*, *PLOP*.

After these steps you can explore the list of PLOP methods under the *PHP Include Path/PLOP/PLOP* node in the PHP Explorer view. When writing new PHP code Eclipse assists with code completion and context-sensitive help for all PLOP methods.

4.8 Python Binding

Installing the PLOP edition for Python. The Python extension mechanism works by loading shared libraries at runtime. For the PLOP binding to work, the Python interpreter must have access to the PLOP Python wrapper which will be searched in the directories listed in the PYTHONPATH environment variable. The name of Python wrapper depends on the platform:

- Unix and macOS: *plop_py.so*
- Windows: *plop_py.pyd*

Error Handling in Python. The Python binding installs a special error handler which translates PLOP errors to native Python exceptions. The Python exceptions can be dealt with by the usual try/except technique:

```
try:
        plop = PLOP()
        ...PLOP instructions...
except PLOPException as ex:
    print("PDFlib PLOP exception occurred:")
    print("[%d] %s: %s" % (ex.errnum, ex.apiname, ex.errmsg))

except Exception as ex:
    print(ex)

finally:
    if plop:
        plop.delete()
```

4.9 Ruby Binding

Installing the PLOP Ruby edition. The Ruby extension mechanism works by loading a shared library at runtime. For the PLOP binding to work, the Ruby interpreter must have access to the PLOP extension library for Ruby. This library (on Windows and Unix: *PLOP.so*; on macOS: *PLOP.bundle)* will usually be installed in the *site_ruby* branch of the local ruby installation directory, i.e. in a directory with a name similar to the following:

```
/usr/local/lib/ruby/site_ruby/<version>/
```

However, Ruby will search other directories for extensions as well. In order to retrieve a list of these directories you can use the following ruby call:

```
ruby -e "puts $:"
```

This list usually includes the current directory, so for testing purposes you can simply place the PLOP extension library and the scripts in the same directory.

Error Handling in Ruby. The Ruby binding installs an error handler which translates PLOP exceptions to native Ruby exceptions. The Ruby exceptions can be dealt with by the usual *rescue* technique:

```
begin
        ...PLOP instructions...
rescue PLOPException => pe
  print "PLOP exception occurred in encrypt sample:\n"
  print "[" + pe.get_errnum.to_s + "] " + pe.get_apiname + ": " + pe.get_errmsg + "\n"
end
```

5 Password Security

5.1 Password Security in PDF

PDF password security offers the following protection features:

- The user password (also referred to as open password) is required to open the file for viewing.
- The master password (also referred to as owner or permissions password) is required to change any security settings, i.e. permissions, user or master password. Files with user and master passwords can be opened for viewing by supplying either password.
- Permission settings restrict certain actions for the PDF document, such as printing or extracting text.
- An attachment password can be specified to encrypt only file attachments, but not the actual contents of the document itself.

If a PDF document uses any of these protection features it is encrypted. In order to display or modify a document's security settings with Acrobat, click *File, Properties..., Security, Show Details...* or *Change Settings...*, respectively. Figure 5.1 shows the security settings dialog in Acrobat.

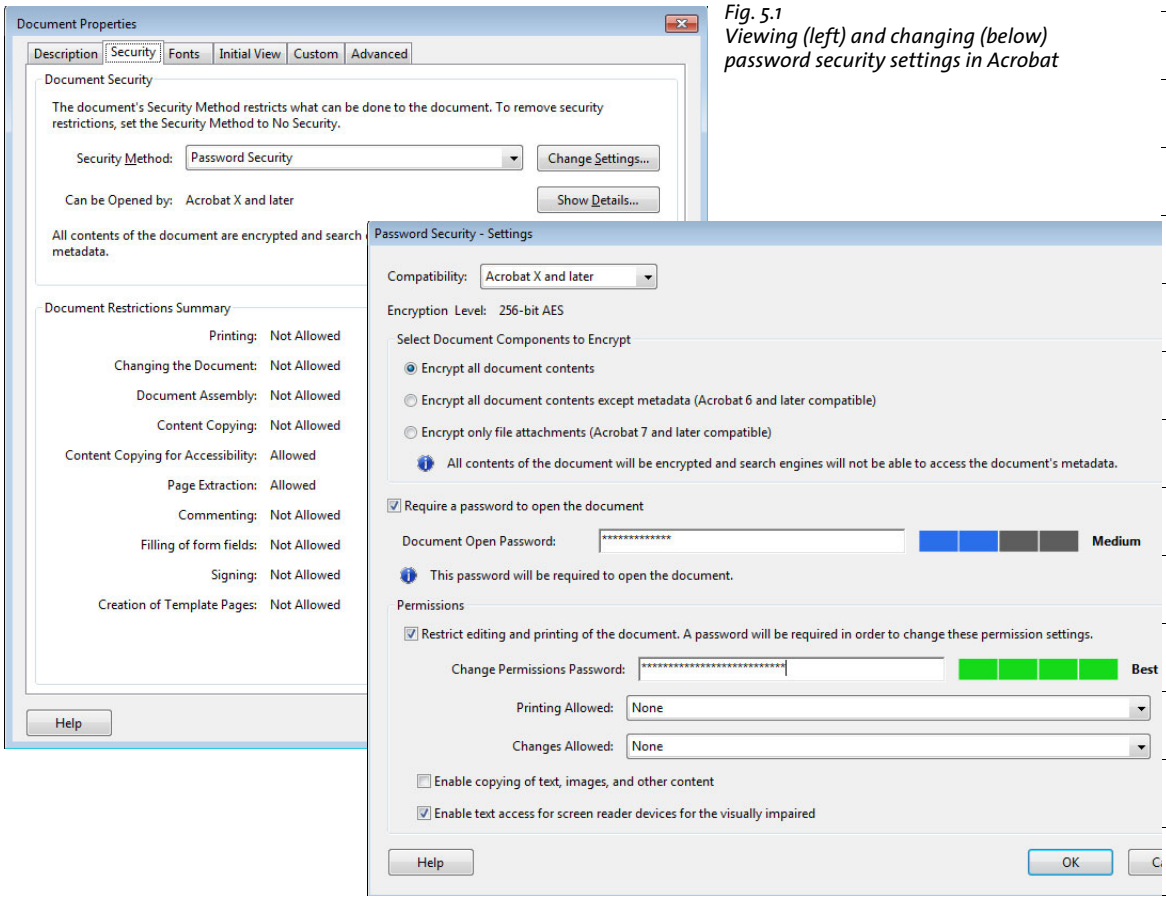


Fig. 5.1
Viewing (left) and changing (below) password security settings in Acrobat

Encryption algorithms and key lengths. PDF encryption makes use of the following encryption algorithms:

- RC4, a symmetric stream cipher (i.e. the same algorithm can be used to encrypt and decrypt). RC4 no longer offers adequate security and has been deprecated in PDF 2.0.
- AES (Advanced Encryption Standard) specified in the standard FIPS-197. AES is a modern block cipher which is used in a variety of applications.

Since the actual encryption keys are unwieldy binary sequences, they are derived from more user-friendly passwords which consist of plain characters. In the course of PDF and Acrobat development the PDF encryption methods have been enhanced to use stronger algorithms, longer encryption keys, and more sophisticated passwords. Table 5.1 details encryption, key and password characteristics for all PDF versions.

Table 5.1 Encryption algorithms, key length, and password length in PDF versions

PDF and Acrobat version, pCOS algorithm number	***encryption algorithm and key length***	***max. password length and password encoding***
PDF 1.1 - 1.3 (Acrobat 2-4), pCOS algorithm 1	*RC4 40-bit (weak; deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.4 (Acrobat 5), pCOS algorithm 2	*RC4 128-bit (weak; deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.5 (Acrobat 6), pCOS algorithm 3	*RC4 128-bit as in PDF 1.4, but different application of encryption method (weak; deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.6 (Acrobat 7) and PDF 1.7 = ISO 32000-1 (Acrobat 8), pCOS algorithm 4	*AES-128 (deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.7ext3 (Acrobat 9), pCOS algorithm 9	*AES-256 with shortcomings in password handling (weak; deprecated in PDF 2.0)*	*127 UTF-8 bytes (Unicode)*
PDF 1.7ext8 (Acrobat X/XI/DC) and PDF 2.0 = ISO 32000-2, pCOS algorithm 11	*AES-256 with improved password handling*	*127 UTF-8 bytes (Unicode)*

Passwords. PDF encryption internally works with encryption keys of 40, 128, or 256 bit depending on the PDF version. The binary encryption key is derived from a password provided by the user. The password is subject to length and encoding constraints:

- Up to PDF 1.7 (ISO 32000-1) passwords were restricted to a maximum length of 32 characters and could contain only characters from the Latin-1 encoding.
- PDF 1.7ext3 introduced Unicode characters and bumped the maximum length to 127 bytes in the UTF-8 representation of the password. Since UTF-8 encodes characters with a variable length of 1-4 bytes the allowed number of Unicode characters in the password is less than 127 if it contains non-ASCII characters. For example, since Japanese characters usually require 3 bytes in UTF-8 representation, up to 42 Japanese characters can be used in passwords.

In order to avoid ambiguities, Unicode passwords are normalized by a process called *SASLprep* (specified in RFC 4013 based on *Stringprep* in RFC 3454). This process eliminates non-text characters and normalizes certain character classes (e.g. non-ASCII space characters are mapped to the ASCII space character U+0020). The password is normalized to Unicode normalization form NFKC, and special bidirectional processing is applied to

avoid ambiguities which may otherwise arise if right-to-left and left-to-right characters are mixed in a password.

The strength of PDF encryption is not only determined by the length of the encryption key, but also by the length and quality of the password. It is widely known that names, plain words, etc. should not be used as passwords since these can easily be guessed or systematically tried using a so-called dictionary attack. Surveys have shown that a significant number of passwords are chosen to be the spouse's or pet's name, the user's birthday, the children's nickname etc., and can therefore easily be guessed.

Permission restrictions. PDF can encode various restrictions on document operations which can be granted or denied individually (see Figure 5.1):

- *Printing Allowed*: If printing is not allowed, the print button in Acrobat will be disabled. Acrobat supports a distinction between *Low Resolution (150 dpi)* and High Resolution printing. Low-resolution printing generates a raster image of the page which is suitable only for personal use, but prevents high-quality reproduction and re-distilling. Note that image-based printing not only results in low output quality, but also considerably slows down the printing process.
- *Changes Allowed:* the corresponding list provides control over various document modification operations:

```
Inserting, deleting, and rotating pages
Filling in form fields and signing existing signature fields
Commenting, filling in form fields, and signing existing signature fields
Any except extracting pages
```

- Content copying is controlled via *Enable copying of text, images, and other content*. While this can be enabled for accessibility with *Enable text access for screen reader devices for the visually impaired*, this setting is considered as deprecated in PDF 2.0 since a PDF reader should always support accessibility.

Specifying access restrictions for a document, such as *Printing Allowed: None* disables the respective function in Acrobat. However, this not necessarily holds true for third-party PDF viewers or other software. It is up to the developer of PDF tools whether or not access permissions are honored. Indeed, several PDF tools are known to ignore permission settings altogether; commercially available PDF cracking tools can be used to disable all access restrictions. This has nothing to do with cracking the encryption; there is simply no way that a PDF file can make sure it won't be printed while it still remains viewable. This is described as follows in ISO 32000-1:

»Once the document has been opened and decrypted successfully, a conforming reader technically has access to the entire contents of the document. There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary.«

Encrypted document components. By default, PDF encryption always covers all components of a document. However, there are use cases where it is desirable to encrypt only some components of the document, but not others:

- PDF 1.5 (Acrobat 6) introduced a feature called plaintext metadata. With this feature encrypted documents can contain unencrypted document XMP metadata. This is for the benefit of search engines which can retrieve document metadata even from encrypted documents.

- Since PDF 1.6 (Acrobat 7) file attachments can be encrypted even in otherwise unprotected documents. This way an unprotected document can be used as a container for confidential attachments.

Security recommendations. The following should be avoided because the resulting encryption is weak and could be cracked:

- Passwords consisting of 1-6 characters should be avoided since they are susceptible to attacks which try all possible passwords (brute-force attack against the password).
- Passwords should not resemble a plain text word since the password would be susceptible to attacks which try all plaintext words (dictionary attack). Passwords should contain non-alphabetic characters. Don't use your spouse's or pet's name, birthday, or other items which are easy to determine.
- The weak RC4 algorithm and AES-256 according to PDF 1.7ext3 (Acrobat 9) should be avoided because it contains a weakness in the password checking algorithm which facilitates brute-force attacks against the password. For this reason Acrobat DC and PLOP never use Acrobat 9 encryption for protecting new documents (only for decrypting existing documents).

In summary, AES-256 according to PDF 1.7ext8/PDF 2.0 should be used. Passwords should be longer than 6 characters and should contain non-alphabetic characters.

Protecting PDFs on the Web. When PDFs are served over the Web users can always produce a local copy of the document with their browser. There is no way for a PDF document to prevent users from saving a local copy.

5.2 Password-protecting PDF Documents with PLOP

PLOP applies or removes standard security features to or from PDF files. PLOP can apply user and master passwords, and set access permissions to prevent printing the document with Acrobat, extracting text, modifying the document, etc. In order to decrypt a document the appropriate master password is required.

Encryption algorithm and key length for password security. PLOP always applies AES-128 (pCOS algorithm 4) or the secure variant of AES-256 (pCOS algorithm 11). PLOP never applies weak RC4 encryption or the weak variant of AES-256 according to PDF 1.7ext3/ Acrobat 9 (pCOS algorithm 9) which contains a weakness in the password handling algorithm. The encryption algorithm can be selected with the *encryption* option of *create_document( ):*

- If *encryption=algo4:* the PDF version is increased to PDF 1.6 if required and AES-128 encryption according to Acrobat 7/8 (pCOS algorithm 4) is applied. Passwords may contain only Latin-1 characters and are truncated to 32 characters.
- If *encryption=algo11* (this is the default): the PDF version is increased to PDF 1.7ext8 if required and AES-256 encryption according to Acrobat X/XI/DC (pCOS algorithm 11) is applied. Passwords may contain Unicode characters and are truncated to 127 UTF-8 bytes.

Required passwords for various PLOP operations. In order to strictly obey the author's intentions as reflected by a PDF document's permission settings, not all operations on documents protected with password security may be allowed. PLOP acts according to the following rules:

- Querying the encryption status with the pCOS pseudo object *encrypt/algorithm* etc. is always possible, regardless of any password.
- Querying document properties with the pCOS interface is governed by the pCOS mode. For example, XMP document metadata, document info fields, bookmarks, and annotation contents can be retrieved without the master password if the document does not require a user password (or only the user password has been supplied). The pCOS Path Reference discusses this in more detail.
- The following operations require the master password: changing or removing the user password, master password, or permission settings, Linearizing, optimizing, repairing, or signing an encrypted document.

Table 5.2 summarizes the requirements for all operations.

Table 5.2 Required passwords for various operations on encrypted documents

known passwords	*query encryption status (pCOS pseudo object »encrypt«)*	*query document info, XMP metadata, bookmarks, annotation contents with pCOS*	*change passwords or permissions, linearize, optimize, repair, or sign*
none	*yes*	*only if no user password is set*	*no*
user	*yes*	*yes*	*no*
master	*yes*	*yes*	*yes*

Setting passwords with PLOP. In the PLOP library API and the PLOP command-line options we refer to the original PDF document as the *input* document, and the encrypted or decrypted result as the *output* document (although both may end up with the same file name). If the input document is protected, PLOP requires either the user or master password depending on the desired operation according to Table 5.2. If the input document could successfully be opened (either because it was unprotected or because the appropriate password has been supplied) any combination of user password, master password, and permission settings can be applied to the output document. However, PLOP interacts with the client-supplied passwords for the output document in the following ways:

- If a user password or permission settings, but no master password has been supplied, a regular user would easily be able to change the security settings, thereby defeating any protection. For this reason PLOP considers this situation as an error.
- If the user and master password are the same, a distinction between user and owner of the file would no longer be possible, again defeating effective protection. PLOP considers this situation as an error.
- Unicode passwords are allowed for AES-256. Older encryption algorithms require passwords which are restricted to the Latin-1 character set. An exception will be thrown for older encryption algorithms if the supplied password contains characters outside the Latin-1 character set.
- Passwords are truncated to 127 UTF-8 bytes for AES-256, and to 32 characters for older encryption algorithms.

Setting permissions with PLOP. PLOP can be used to query, set or remove any of the permission settings detailed in Table 5.3. Unless specified otherwise, all actions are allowed by default. Specifying access restrictions disables the respective feature in Acrobat. Access restrictions can be applied without setting a user password, but a master password is required. Table 5.3 lists the supported permission restriction keywords.

Table 5.3 Permission restriction keywords for the `permissions` *option of create_document( ) and add_recipient( )*

keyword	*explanation*
Printing the document	
nohiresprint	*Prevent high-resolution printing. If* `noprint` *hasn't been specified, printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.*
noprint	*Prevent printing the file.*
Changing the document	
nomodify	*Prevent adding form fields or making any other changes.*
noannots	*Prevent adding or changing comments and filling in form fields. If* `nomodify` *and* `noannots` *haven't been specified, creating and modifying form fields (including signature fields) is allowed.*
noforms	*(Implies* `noannots`*) Prevent form field filling and signing, even if* `noannots` *hasn't been specified.*
noassemble	*(Implies* `nomodify`*) Prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if* `nomodify` *hasn't been specified.*
Copy contents from the document	
nocopy	*Prevent copying and extracting text or graphics.*
noaccessible	*(Deprecated in PDF 2.0) Prevent extracting text or graphics for accessibility purposes.*

Table 5.3 Permission restriction keywords for the `permissions` *option of create_document( ) and add_recipient( )*

keyword	***explanation***
other	
plainmetadata	*(Only for create_document( )) Keep document metadata unencrypted even for encrypted documents.*
nomaster	*(Only for add_recipient( )) Restrict printing, editing, and content extraction according to the specified permission restriction keywords, and prevent changing the document's security settings. If this keyword is not supplied, the recipient has full rights to the document, i.e. all other permission restriction keywords (except* `plainmetadata`*) are ignored.*

Note that Acrobat doesn't provide full control over all four permission restrictions related to changing the document, but groups some of the restrictions together. Table 5.4 contains a comparison of Acrobat settings (the values of the »Changes allowed« list in Figure 5.1) and corresponding PLOP permission restriction keywords.

Table 5.4 Permission descriptions in Acrobat and corresponding keyword combinations for the `permissions` *option of create_document( ) and add_recipient( )*

»Changes allowed« in Acrobat	***corresponding permissions keywords***
None	`nomodify noannots noforms noassemble`
Inserting, deleting, and rotating pages	`nomodify noannots noforms`
Filling in form fields and signing existing signature fields	`nomodify noannots noassemble`
Commenting, filling in form fields, and signing existing signature fields	`nomodify noassemble`
Any except extracting pages	`noassemble`

5.3 Applying Password Security on the Command-Line

You can encrypt documents by specifying the *userpassword* or *masterpassword* option (or both) for *create_document( )*. Note that a user password always requires a master password, but not vice versa. Full sample code for securing PDF documents and removing security with the PLOP library can be seen in the *encrypt* and *decrypt* programming samples, which are included in all PLOP packages. The equivalent options for the PLOP command-line tool are *--user* and *--master*.

Permission restrictions can be specified with the *permissions* option for *create_document( )*; the equivalent option for the command-line tool is *--permissions*.

Note On Windows passwords on the command line may contain Unicode characters outside the Latin-1 character set.

Encryption examples. The sample command-line calls below are shown with long command-line options; see Section 3.1, »PLOP and PLOP DS Command-line Options«, page 39, for abbreviated options.

Encrypt a file with user password *demo* and master password *DEMO*:

```
plop --user demo --master DEMO --outfile encrypted.pdf input.pdf
```

Encrypt all files in the current directory with the same user password *demo* and master password *DEMO*, and place the resulting files in the target directory *output:*

```
plop --targetdir output --user demo --master DEMO *.pdf
```

Passwords which contain space characters must be enclosed in braces (to follow option list syntax) and with straight quote characters (to follow shell syntax) as in the following example: encrypt a document with the master password *two words*:

```
plop --master "{two words}" --outfile encrypted.pdf input.pdf
```

Encrypt and sign a document (see Section 7.2.2, »Signing with the built-in Engine«, page 94, regarding the signature options):

```
plop --user demo --master DEMO --outfile signed+encrypted.pdf ↵
        --signopt "update=false digitalid={filename=demo_signer_rsa_2048.p12} ↵
        password=demo" input.pdf
```

Permission settings. Apply the master password *DEMO* and the permission settings *noprint, nocopy,* and *noannots* to all files in a directory, and copy the resulting files to the target directory *output*. AES encryption is applied regardless of the encryption used in the input documents. Verbosity level 2 prints the names of all input and output files as they are processed:

```
plop --verbose 2 --master DEMO ↵
        --permissions "noprint nocopy noannots" --targetdir output *.pdf
```

Remove all permission restrictions from a file, and copy the result to a different output file with the same master password. This requires the master password for the input document:

```
plop --password DEMO --master DEMO --outfile unrestricted.pdf protected.pdf
```

Re-encrypt a document (e.g. to replace weak encryption with strong AES encryption or weak passwords with better ones), and clone the permission settings of the input document. Copy the result to a different output file. This requires the master password for the input document:

```
plop --password DEMO --master LONGPASSWORD --permissions keep ↵
       --outfile unrestricted.pdf protected.pdf
```

Decryption examples. Decrypt a single file with the master password *DEMO*. All access restrictions which may have been applied to the input document will be removed (since the output is unencrypted):

```
plop --password DEMO --outfile decrypted.pdf encrypted.pdf
```

Re-encrypt with stronger crypto. PLOP can be used to apply stronger encryption to documents which are encrypted with short keys or weak passwords. You must supply the old and the new password. PLOP uses strong AES encryption by default. The following example assumes that the input is encrypted with the master password *old*, and the output will be AES-encrypted with the master password *DEMO*. The new password can even be the same as the old password. Of course you should only use really strong passwords (see »Security recommendations«, page 64), not short ones as in this example:

```
plop --password old --master DEMO --outfile strong.pdf weak.pdf
```

6 Certificate Security

6.1 Certificate Security in Acrobat

Advantages of certificate security. PDF documents which are protected with password security can be opened if the user or master password is known. The disadvantage is that password distribution may be difficult since it requires a confidential channel. Also, legitimate document recipients could accidentally or deliberately share passwords with other parties.

Certificate security offers an alternative to password security. It is based on public key cryptography and certificates. A document is encrypted for a number of recipients, where each recipient is identified by his certificate. Since certificates contain only the public key, but no confidential information, they don't require any protection and can freely be distributed. In order to open a protected document a recipient needs the digital ID with the private key corresponding to the certificate which has been used for encryption.

Certificate security offers the following advantages over password security:

- No passwords must be distributed to the recipients.
- Individual permission restrictions can be specified for each recipient or group of recipients. Permissions are useful for distributing documents to users with different usage rights.

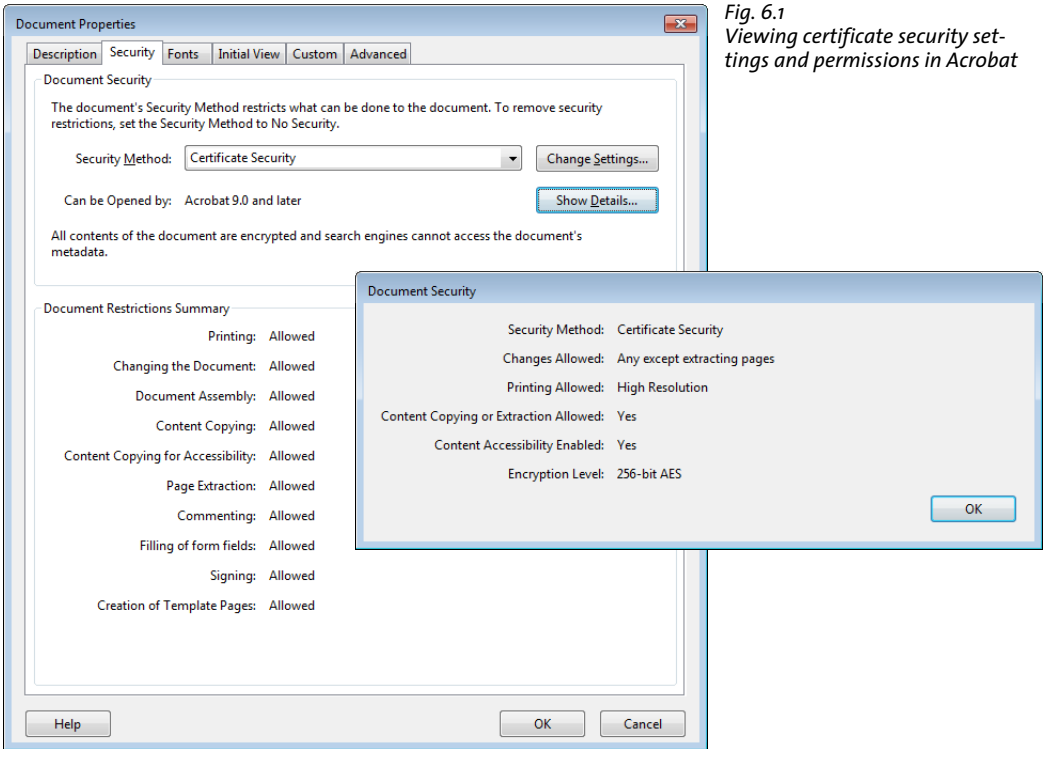


Fig. 6.1
Viewing certificate security settings and permissions in Acrobat

- Recipients cannot pass on document passwords to unauthorized third parties. While they could copy and pass on their digital software ID, the ID reveals their name and could further be abused, e.g. for forging their signatures. Also, hardware-based digital IDs cannot be copied.

Certificate Security is supported in Acrobat and Adobe Reader 6 and above. In the sections below we provide an overview of certificate security in Acrobat. Please refer to the Acrobat documentation for more details.

Preparations for certificate security. In order to work with certificate security you need digital certificates. More precisely, you need a digital ID (with a public/private key pair) for yourself and certificates (containing only a public key) for each recipient. There are two main options for obtaining certificates:

- Self-signed, e.g. created with Acrobat: if you receive certificates directly from the recipients this method is simple and available at no additional cost. However, when a digital ID is lost it cannot be recovered, which means encrypted documents can no longer be opened.
- Digital IDs from a commercial CA are available for a fee, but they can be recovered in case of loss. If AATL certificates are used they can also be used for digitally signing documents such that validation in Acrobat doesn't require any extra configuration (see Section 7.1.3, »Trusted Root Certificates in Acrobat«, page 90).

For creating protected documents you need only the recipients' certificates with the public key. This differs from the requirements for opening protected documents where each recipient including yourself needs the corresponding digital ID with the private key. Since certificates don't contain any confidential information they don't require a password and can freely be distributed, while digital IDs are usually protected with a password or PIN.

Configuring your own digital ID for decryption. Acrobat supports several methods for configuring a digital ID so that it can be used for opening a protected document. Proceed as follows to create or install your own digital ID with Acrobat DC: *Edit, Preferences..., Signatures, Identities & Trusted Certificates, More..., Digital IDs, Add ID*.
In the resulting dialog *Add Digital ID* you can either add an existing ID from file or create a new self-signed ID.

In order to export a trusted certificate or to create a certificate for your own digital ID (so that others can encrypt documents for you) proceed as follows with Acrobat DC: *Edit, Preferences..., Signatures, Identities & Trusted Certificates, More..., Trusted Certificates* (for other people's certificate) or *Digital IDs* (for your own certificate), select the ID or certificate, *Certificate Details*.
This brings up the Certificate Viewer where you click *Export...* and *Save the exported data to a file: Certificate File* (not *Certificate Message Syntax - PKCS#7).* The exported certificate can be used as a recipient certificate (provided it is not yet expired).

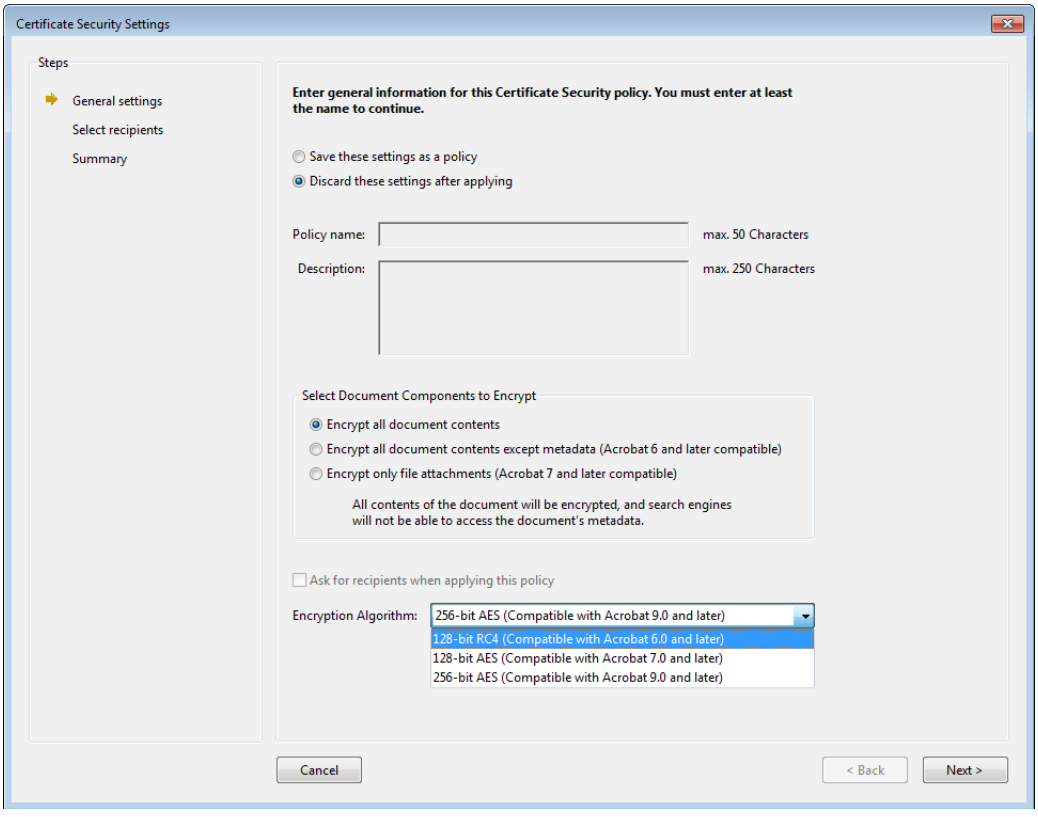
Note Acrobat can also use digital IDs in the Windows certificate store.

Applying certificate security with Acrobat. Once you configured your own digital ID and obtained recipient certificates you can encrypt a PDF document with the following steps in Acrobat DC:

- Click *File, Properties..., Security* tab, *Security Method: Certificate Security*. In Acrobat DC you can use the following alternative sequence: *Tools, Protect, Encrypt, Encrypt with Certificate*. In the resulting dialog you can specify the type of PDF encryption which is used for certificate security (see Figure 6.2).
- In the next dialog you should select your own digital ID to ensure that you will be able to open the encrypted document later.
- Now select an arbitrary number of recipient certificates and adjust permissions if desired. The recipient certificates can be pulled from the Windows certificate store, read from a file, or fetched from an online repository.
- Saving the file encrypts it according to the selected security settings.

In order to open the encrypted document again you need a digital ID with the private key which corresponds to one of the recipient certificates. The ID can be installed in Acrobat's certificate store or in the Windows certificate store.

Fig. 6.2
Acrobat's certificate security dialog with choice of PDF encryption algorithms at the bottom



Decryption errors. If Acrobat cannot decrypt a document it issues the following error message:

A digital ID was used to encrypt this document but no digital ID is present to decrypt it. Make sure your digital ID is properly installed or contact the document author.

However, this message appears not only if the corresponding digital ID is missing, but also if the document cannot be decrypted for other reasons.

Acrobat incompatibility with ECC recipient certificates. Acrobat DC supports Elliptic Curve Cryptography (ECC) with curves P-256/P-384/P-521 and the other curves recommended by NIST for digital signatures and encryption. However, Acrobat creates a CMS encryption object which doesn't conform to RFC 5652 as amended by RFC 5753 *»Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)«*. This makes encrypted documents incompatible with third-party software.

6.2 Certificate Security in PDF

6.2.1 CMS Enveloped Data

PDF certificate security is based on the *Cryptographic Message Syntax* (CMS) according to RFC 5652. CMS describes an encapsulation syntax for various cryptographic features including digital signatures, message authentication and encryption. PDF certificate security makes use of the *Enveloped Data* functionality provided by CMS; encrypted e-mail works in a similar manner. To optimize storage requirements and performance certificate security is implemented in a hybrid manner: first an encryption key is created randomly and encrypted against each recipient's certificate. This process then uses public key encryption based on the RSA or Elliptic Curve Cryptography (ECC) algorithms and stores the encrypted versions of the random key in the CMS object. The random key is used to encrypt the actual CMS payload with a symmetric algorithm and store it in the CMS. Modern implementations generally use the AES algorithm for symmetric encryption. See Section 6.2.2, »Cryptographic Details«, page 77, for more details on this process and the involved algorithms.

Each recipient uses his private key to decrypt the symmetric key, and then uses the resulting key to decrypt the CMS payload. The combination of encrypted payload and an encrypted content encryption key for each recipient is called a digital envelope.

Since the first encryption step is asymmetric and the random key is kept only temporarily, the creator of an encrypted document cannot decrypt it later unless his own certificate is included in the list of recipients.

There are two reasons for the hybrid approach with asymmetric and symmetric encryption. Firstly, asymmetric encryption is very slow and suited only for small amounts of data. It is therefore applied only to the short symmetric encryption key and not the full payload. Secondly, this approach allows the payload to be encrypted only once with symmetric encryption, while only the short encryption key must be encrypted individually for each recipient. Encrypting the payload for each recipient would significantly increase the output file size.

The *EnvelopedData* structure in a CMS object contains one or more *RecipientInfo* structures (see Figure 6.4). Each of these contains information about a recipient's certificate – usually the certificate issuer (CA) and serial number – and an encrypted key for the recipient.

Recipient confidentiality. The recipient names are not present in the CMS object. However, the name of the certificate issuer as well as the certificate's serial number can be retrieved from the CMS object without decryption. For certificates retrieved from a Public Key Infrastructure this means that only the name of the CA and the serial number are exposed. Depending on the PKI this information may or may not be sufficient for identifying the recipient. However, for self-signed certificates the certificate holder herself signs the public key. As a result, the names of all recipients with self-signed certificates are visible in plaintext in the CMS object. In some situations exposing recipient information may be undesirable. You can solve this problem by avoiding self-signed certificates, or by using pseudonyms in self-signed certificates.

Number of recipients and CMS size. A document can be encrypted against an arbitrary number of recipient certificates. However, since a uniquely encrypted key is embedded for each additional recipient, the size of the CMS increases with the number of recipi-

ents. The file size expansion depends on the length of the recipient's public key and the amount of information in the certificate. Typically, the output file size increases by ca. 1-2 KB per recipient.

Applying CMS to PDF Documents. A PDF document encrypted with certificate security contains one or more CMS object in the *Recipients* entry of the *Encrypt* dictionary. However, PDF doesn't apply the CMS mechanism directly to the document contents, but adds another layer of encryption which is identical to password security. The CMS payload doesn't contain any PDF objects, but keying material which is used to derive the encryption key for PDF objects. The symmetric algorithms used for encrypting PDF objects are the same which are used for password security (see Table 5.1). While certificate security derives the document encryption key from the encrypted keying material in the CMS, password security derives this key from the secret password.

PDF permission restrictions. The same permission restrictions as for password security can be applied to documents protected with certificate security (see Figure 6.1 and »Permission restrictions«, page 63). In addition to the permission restrictions for password security the following setting is possible with certificate security:

- *Restrict printing and editing of the document and its security settings* (see top of Figure 6.3): If this restriction is active, the recipient can open and read the document, but certain operations such as printing and modifying the document are governed by other permission restrictions. If this restriction is not active, the recipient has full control over the document and can also change the security settings. This is similar to documents protected with password security where the user knows the master password. For this reason we call this *master permission*.

Different users may be assigned different permission restrictions. For example, in an enterprise context a manager may be assigned master permission so that she can edit the document, change the encryption or apply any other change, while her coworkers are only allowed to fill form fields and sign the document. The ability to apply specific permissions to each recipient or group of recipients is an important advantage of certificate security over password security.

The permissions are included in the CMS payload which is encrypted for a particular recipient. For this reason permission restrictions for a specific recipient can only be determined upon decryption for this recipient. It is not possible to query a recipient's permissions without access to the corresponding digital ID.

Permission handling is a noteworthy difference between certificate security for PDF documents and e-mail. A PDF document can contain multiple CMS objects, where each

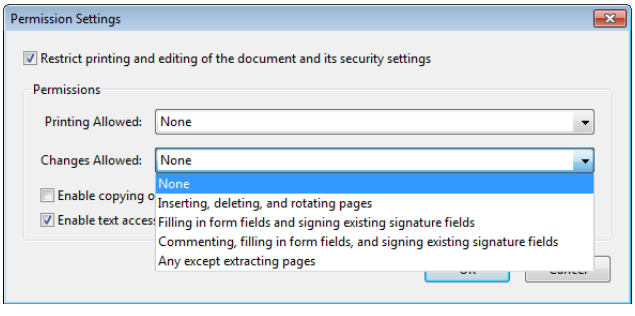


Fig. 6.3
Setting permission restrictions for certificate security in Acrobat

object may in turn address a group of several recipients. Encrypted keys for recipients with the same permissions are stored in the same CMS object.

6.2.2 Cryptographic Details

Certificate security involves multiple encryption steps which may use different algorithms and key lengths. The encryption steps described below are noted in Figure 6.4.

Step 1: CMS public key encryption and key wrap. Public key encryption is deployed to encrypt a randomly generated content encryption key (CEK), where the details vary between RSA and ECC recipient certificates. Different recipients may use a mixture of RSA and ECC keys as well as different RSA key lengths or ECC curves.

If the recipient's certificate contains a public key for the RSA algorithm (RFC 5652), this key is used to encrypt the content encryption key. RSA encryption requires a padding method. The default PKCS#1 v1.5 method is supported in all Acrobat versions. The newer OAEP (Optimal asymmetric encryption padding) according to PKCS#1 v2 (identical with RFC 3447) and RFC 3560 offers security advantages; it can be requested with the option *rsapadding=oaep*. OAEP is not supported in Acrobat DC and below, but in some third-party PDF viewers.

If the recipient's certificate contains a public ECC key (RFC 5753), the Elliptic Curve Diffie-Hellman (ECDH) key agreement scheme and the public key in the recipient certificate are used to derive yet another temporary key encryption key. A symmetric encryp-

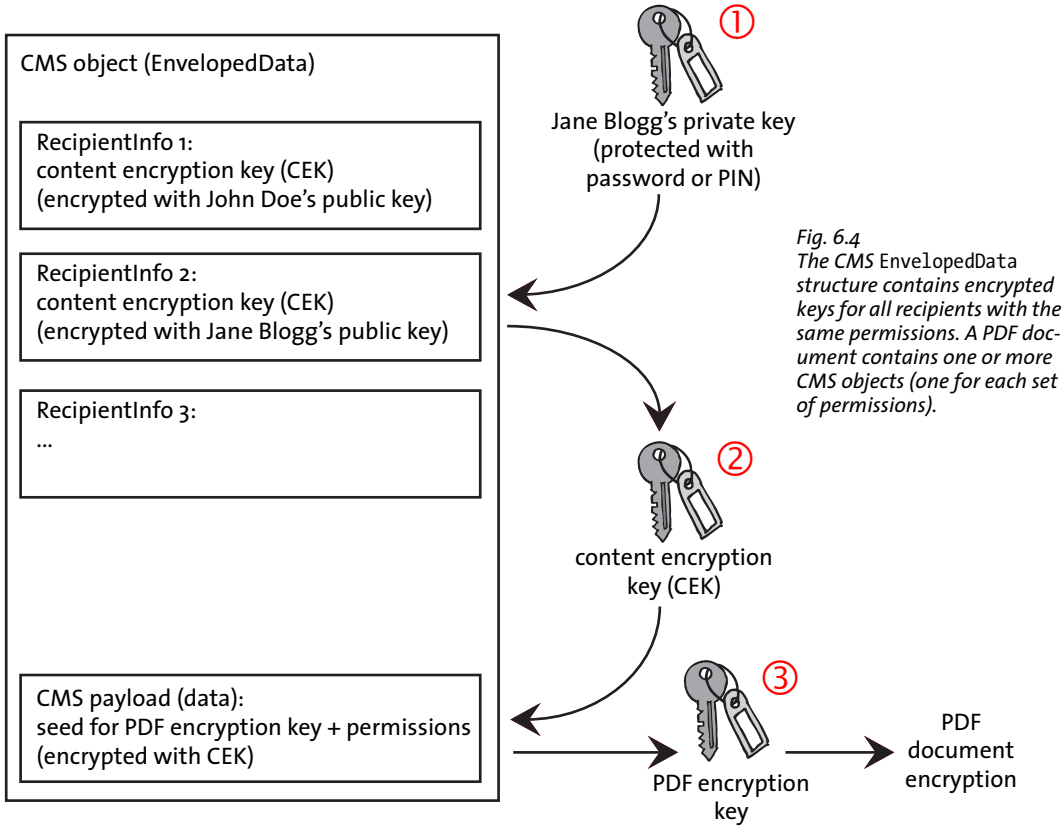


Fig. 6.4
The CMS `EnvelopedData` *structure contains encrypted keys for all recipients with the same permissions. A PDF document contains one or more CMS objects (one for each set of permissions).*

tion algorithm called key wrap algorithm is then used to encrypt (wrap) the content encryption key with the key encryption key. Acrobat XI/DC uses AES-128 or AES-256 as key wrap algorithm.

Step 2: CMS content encryption. The content encryption key is used to encrypt the PDF encryption keying material (not the actual key itself) with a symmetric algorithm, resulting in the encrypted CMS payload. With PDF certificate security the CMS »content« doesn't contain any PDF document data, but instead some keying material from which the final encryption key for PDF objects is derived.

Acrobat XI/DC uses AES-128 or AES-256 for content encryption. Since payload encryption is required only once regardless of the number of recipients, the choice of algorithm doesn't depend on the recipient certificates.

Step 3: PDF encryption. The PDF encryption key is applied to PDF objects which results in the data for displaying the document. This step is identical to password security.

The symmetric algorithm and key length for encrypting PDF objects are specified in the PDF Reference and correspond to a subset of those used for password security (see Table 5.1). The same symmetric algorithm is used to encrypt all objects in the PDF document. Only the algorithm for step 3 can be selected in Acrobat (see Figure 6.2, bottom). For the strong encryption introduced with Acrobat 7 and Acrobat 9 the PDF encryption algorithm is AES-128 or AES-256, respectively.

Algorithms and key lengths. Table 6.1 summarizes algorithms and key lengths for various PDF and Acrobat versions.

Table 6.1 PDF and CMS encryption algorithms for certificate security and support in Acrobat

PDF/Acrobat version and pCOS algorithm number	***Step 1: CMS public key algorithm***	***Step 2: CMS content encryption***	***Step 3: PDF encryption***
PDF 1.4 (Acrobat 5), pCOS algorithm 5	*Acrobat 6 and above: 2048-bit RSA*	*Acrobat XI/DC: 128-bit AES* *PLOP: only reading supported*	*128-bit RC4 (weak; deprecated in PDF 2.0)*
PDF 1.5 (Acrobat 6), pCOS algorithm 12	*Acrobat 6 and above: up to 2048-bit RSA*	*Acrobat XI/DC: 128-bit AES* *PLOP: only reading supported*	*128-bit RC4 (weak; deprecated in PDF 2.0)*
PDF 1.6 (Acrobat 7), pCOS algorithm 6	*Acrobat 8 and above: up to 8192-bit RSA*	*Acrobat XI/DC: 128-bit AES* *PLOP: 128-bit AES*	*128-bit AES (deprecated in PDF 2.0)*
PDF 1.7ext3 (Acrobat 9), pCOS algorithm 10	*Acrobat 9 and above: up to 8192-bit RSA*[1] *Acrobat DC: ECC with curve P-256/P-384/P-521*[2]	*Acrobat XI/DC: 256-bit AES* *PLOP: 256-bit AES*	*256-bit AES (strong)*

1. RSA with OAEP padding is not supported in Acrobat DC and below.
2. The key derivation function `dhSinglePass-stdDH-sha512kdf` *(optional in RFC 5753) is not supported in MSCAPI. Therefore such documents cannot be decrypted by PLOP with* `engine=mscapi`*, but only with* `engine=builtin`*.*

6.3 Use Cases for Certificate Security

In this section we discuss use cases which benefit from the advantages of certificate security. In most cases the following questions should be analyzed:

- Is it required to include the author's own certificate in the list of recipients? If it is not included the author won't be able to open the protected document.
- Which permission restrictions are applicable to each recipient or group of recipients?

Distribute confidential documents to a closed group of recipients. Members of a group want to exchange confidential documents so that all other group members can use the documents. The PDFs are encrypted against the certificates of all group members. If the creator of a document includes his own certificate when encrypting the file, only a single version of the document is required. Although the number of recipients is not strictly limited, it should be kept in mind that each recipient slightly enlarges the document.

In a variation of this use case some recipients (the managers) are allowed to modify the document, while regular employees are only allowed fill form fields and sign the PDF. This distinction can be achieved with separate recipient groups where each group is assigned appropriate permissions.

If the number of group members gets large (thousands of recipients), the group can be split into smaller sets. A small number of recipients in each subset minimizes the file size, while a large number of recipients in each subset reduces the number of different protected versions which must be created based on the same document.

Serially signing a confidential document. A confidential document is encrypted for a number of recipients. The recipients are expected to digitally sign the document, but are not allowed to apply any modification. To achieve this, the permissions are set to allow only signing, but no modifications. Recipients can use the same digital ID for decrypting and signing the document, provided it has been issued to allow both activities.

Digital rights management. Documents with commercial contents are distributed to paying customers. Each subscriber or buyer receives a protected PDF which has been encrypted against his personal certificate. In order to create an individual document version for each recipient, many protected versions of the same document can be created. The *nomaster* permission restriction is set to prevent customers from tampering with the document.

Secure storage and archiving. In this scenario an archive receives documents which must be protected. Each archived document is encrypted against the archive owner's certificate. Only a single protected version must be created for each archived document.

Invoice and statement distribution. A customer-specific invoice, statement or transaction document is encrypted against the customer's certificate to ensure confidentiality. A single protected version of each document is created and provided to the customer. The *nomaster* permission restriction is set to prevent customers from tampering with the document.

6.4 Certificate Security with PLOP

PDF encryption algorithm and key length. PLOP always applies certificate encryption in combination with the strong algorithms AES-128 or AES-256, but never the weak RC4 algorithm. The PDF encryption algorithm can be selected with the *encryption* option of *create_document( ):*

- For *encryption=algo6:* the PDF version is increased to PDF 1.6 if required and certificate encryption with AES-128 according to Acrobat 7 (pCOS algorithm 6) is applied.
- For *encryption=algo10* (which is the default): the PDF version is increased to PDF 1.7ext3 if required and certificate encryption with AES-256 according to Acrobat 9 (pCOS algorithm 10) is used.

The PDF encryption algorithm (i.e. AES-128 or AES-256) is also used for CMS content encryption; no weak algorithms are deployed.

Specifying recipient certificates. For each recipient a certificate must be supplied which contains the recipient's public key. Unlike a digital ID a certificate does not contain any private key and therefore doesn't have to be protected. The PDF document is encrypted such that only the specified recipients are able to decrypt it with the private key corresponding to the public key in their certificate.

Each recipient must be specified with a call to *add_recipient( )* before an output document can be created (full sample code is available in the *certsec* mini sample which is included in all PLOP packages):

```
if (plop.add_recipient("certificate={filename=demo_recipient_1.pem}") == -1)
{
        /* emit warning and continue */
        System.err.print("Warning: ", plop.get_errmsg());
}
...
if (plop.create_document(out_filename, optlist) == -1) {
        System.err.println("Error: " + plop.get_errmsg());
        plop.delete();
        System.exit(2);
}
```

Specifying at least one recipient activates certificate security. Once a list of recipients has been created, it will be applied to all subsequently generated documents until a new list is created with additional calls to *add_recipient( )*.

Recipients can be specified in the PLOP command-line tool with the *--recipient* option.

Requirements for recipient certificates. A recipient certificate used for protecting PDF documents with PLOP must meet the following requirements:

- If the certificate contains a key usage extension it must enable the certificate for encryption or key agreement. Certificates which are only enabled for digital signatures cannot be used for encryption.
- The certificate must be valid, i.e. its expiration date must not have been reached.
- By default RSA keys are only accepted if the key length is a multiple of 8 bits. Odd-sized keys are accepted with the option *conformance=extended*. However, Acrobat cannot open the resulting documents with a digital ID in the Windows certificate

store, but only with an ID in the Acrobat certificate store. It is recommended to use keys where the length is a multiple of 64 bits.

- ECC recipient certificates with curves other than P-256/P-384/P-521 are rejected by default, but are accepted with the option *conformance=extended*. However, the encrypted documents cannot be opened with Acrobat XI/DC.

Permission restrictions. Permission restrictions, e.g. printing not allowed, can be specified separately for each recipient in the *permissions* option of *add_recipient()*. Unless specified otherwise, all actions are allowed by default. Table 5.3 lists the supported permission restriction keywords.

Note that Acrobat doesn't provide independent control over all four permission restrictions related to changing the document, but groups some of the restrictions together. Table 5.4 relates Acrobat settings (the values of the »Changes allowed« list in Figure 6.3) to the corresponding combinations of PLOP permission restriction keywords.

Sample option list for *add_recipient()* to prohibit all document modifications except form filling and signing:

```
certificate={filename=demo_recipient_1.pem} permissions={nomodify noannots noassemble}
```

Cryptographic engines. The *builtin* and *mscapi* engines discussed in Section 7.2, »Signing with PLOP DS«, page 93, can also be used for retrieving recipient certificates for encryption. The *pkcs#11* engine is not supported for certificate security. In order to protect documents against a certificate on a token you can use the token vendor's administration software to extract the certificate from the token and supply it as a software certificate with the *builtin* engine. This approach doesn't reduce security since the certificate contains only public information which (unlike the private key on the token) doesn't have to be kept confidential.

The engine can be specified with the *engine* option of *add_recipient()* and determines suboptions for certificate selection:

- With *engine=builtin* (the default) certificates must be supplied as X.509 files in PEM or DER encoding, e.g.

  ```
  engine=builtin certificate={filename=demo_recipient_1.pem}
  ```

  The specified certificate file must contain exactly one encryption certificate.
- With *engine=mscapi* certificates can be fetched from the Windows certificate store. They are selected by the name of the certificate store and the recipient's subject name in the certificate, e.g.

  ```
  engine=mscapi certificate={store=My subject={PLOP Demo Recipient 1}}
  ```

Decrypting protected documents. In order to decrypt a document which is protected with certificate security you need a digital ID corresponding to one of the recipient certificates in the document. The *digitalid* option of *open_document()* must be supplied along with the corresponding password for accessing the ID, e.g.

```
digitalid={filename=demo_recipient_1.p12} password=demo
```

If the digital ID is fetched from the Windows certificate store, all IDs in the specified store are checked to decrypt the document. For this reason the *subject* suboption for selecting an ID is not required:

```
engine=mscapi digitalid={store=My}
```

Since *My* is the default store name this can be further abbreviated as follows:

```
engine=mscapi
```

Required credentials for various PLOP operations. In order to strictly obey the author's intentions as reflected by a PDF document's permission settings, not all operations on documents protected with certificate security may be allowed. PLOP processes documents which are protected with certificate security according to the following rules:

- Querying the encryption status with the pCOS pseudo object *encrypt/algorithm* etc. is always possible, regardless of the availability of a suitable digital ID.
- Querying other document properties with the pCOS interface requires a suitable digital ID, i.e. an ID with a private key which matches one of the recipient public keys in the encrypted document. This can be checked with *pcosmode=1* or *pcosmodename= restricted*.
- Signing a document in incremental update mode also requires a suitable digital ID. In addition the *noannots* permission setting in the document must be set to *false* to allow signing. This can be checked with the *encrypt/noannots* pCOS pseudo object.
- Processing the document in any other way, e.g. removing encryption or changing the permission settings, requires a suitable digital ID. In addition the document must set master permission for the ID which is used to open the document. This can be checked with *pcosmode=2* or *pcosmodename= full*.

Table 6.2 summarizes the requirements for all operations.

Table 6.2 Required digital IDs for various operations on encrypted documents

available ID, master permission and pCOS mode	*query encryption status with pCOS*	*query other document properties with pCOS*	*sign in update mode*	*other processing, e.g. change encrypt.*
none (pCOS mode 0/minimum)	*yes*	*no*	*no*	*no*
suitable ID available and master permission not set (pCOS mode 1/restricted)	*yes*	*yes*	*only if* `noannots=false`	*no*
suitable ID available and master permission set for this ID (pCOS mode 2/full)	*yes*	*yes*	*yes*	*yes*

Querying recipients and permissions with pCOS. You can use the pCOS pseudo object *encrypt/recipients* to check for certificate security. If the value of

```
length:encrypt/recipients
```

is larger than zero, each entry in this array contains a CMS object for a group of one or more recipients with identical permissions. Since each CMS object may contain one or more recipients the array length does not necessarily indicate the total number of recipients.

The *pcosmode* and *pcosmodename* pseudo objects can be used to check whether a suitable digital ID was supplied for opening the document, and whether the master permission is set:

- Minimum pCOS mode *(pcosmode=0* or *pcosmodename= minimum):* no suitable digital ID was supplied.
- Restricted pCOS mode *(pcosmode=1* or *pcosmodename= restricted):* a suitable digital ID for opening was supplied, but the document does not set master permission for this recipient. Signing the document is only allowed if the *noannots* permission is set to *false*.
- Full pCOS mode *(pcosmode=2* or *pcosmodename= full):* a suitable digital ID was supplied and the document sets master permission for this recipient. All document permissions are granted without restriction and all PLOP operations are allowed.

Permission restrictions can be checked with the following entries in the *encrypt* pCOS pseudo object (e.g. *encrypt/noassemble):*

```
noaccessible, noannots, noassemble, nocopy, noforms, nohiresprint, nomodify, noprint
```

Note that there is no pseudo object *encrypt/nomaster* since the status of the master permission flag can be checked with *pcosmode=2* or *pcosmodename= full*. Refer to the pCOS Path Reference for more information. The *dumper* mini sample contains code for identifying documents with certificate security.

6.5 Applying Certificate Security on the Command-Line

The sample command-line calls below are shown with long command-line options; see Section 3.1, »PLOP and PLOP DS Command-line Options«, page 39, for abbreviated options.

Encryption. Recipient certificates can be specified in the *--recipient* command-line option or its short form *-r*. This option can be repeated for multiple recipients.

Encrypt a document for a single recipient provided the certificate is available on file:

```
plop --recipient "certificate={filename=demo_recipient_1.pem}" ↵
        --outfile encrypted.pdf input.pdf
```

Encrypt a document for a recipient and restrict its permissions so that printing and copying are not allowed:

```
plop --recipient "certificate={filename=demo_recipient_1.pem permissions={noprint
        nocopy}}" --outfile encrypted.pdf input.pdf
```

Encrypt a document for two recipients, provided the certificates are available on file:

```
plop --recipient "certificate={filename=demo_recipient_1.pem}" ↵
     --recipient "certificate={filename=demo_recipient_2.pem}" ↵
     --outfile encrypted.pdf input.pdf
```

Encrypt a document for a large number of recipients: in this situation response files for the PLOP command-line tool are useful (see »Response files«, page 41). Create a text file *recipients.txt* with all required recipient options:

```
--recipient "certificate={filename=demo_recipient_1.pem}"
--recipient "certificate={filename=demo_recipient_2.pem}"
--recipient "certificate={filename=demo_recipient_3.pem}"
--recipient "certificate={filename=demo_recipient_4.pem}"
...
```

Next, supply the name of this response file in the PLOP command-line call, preceded by an @ character:

```
plop @recipients.txt --outfile encrypted.pdf input.pdf
```

Encrypt a document for a single recipient for which a certificate is available in the Windows certificate store:

```
plop --recipient "engine=mscapi certificate={store=My subject={PLOP Demo Recipient 1}}" ↵
     --outfile encrypted.pdf input.pdf
```

Encrypt and sign a document (see Section 7.2.2, »Signing with the built-in Engine«, page 94, regarding the signature options). This requires the recipient certificate and the signer's digital ID:

```
plop --recipient "certificate={filename=demo_recipient_1.pem}" ↵
     --signopt "update=false digitalid={filename=demo_signer_rsa_2048.p12} password=demo" ↵
     --outfile signed+encrypted.pdf input.pdf
```

Permission settings. Protect a document for two recipients where the first recipient is granted full access, while the second recipient is only allowed to sign the document without applying any changes. Since the *noforms* permission restriction keyword is absent, form filling and signing are allowed for the second recipient:

```
plop --recipient "certificate={filename=demo_recipient_1.pem}" ↵
     --recipient "certificate={filename=demo_recipient_2.pem} ↵
        permissions={nomodify nocopy noannots noassemble}" ↵
     --outfile encrypted.pdf input.pdf
```

Encrypt a document for a single recipient who is only allowed to view the document:

```
plop --recipient "certificate={filename=demo_recipient_1.pem} ↵
        permissions={noprint nomodify nocopy noannots noassemble noforms}" ↵
     --outfile encrypted.pdf input.pdf
```

Decryption. Decrypt a document which has been protected with certificate security and create an unprotected version, assuming a suitable digital ID is available on file:

```
plop --inputopt "digitalid={filename=demo_recipient_1.p12} password=demo" ↵
     --outfile decrypted.pdf encrypted.pdf
```

Decrypt a protected document and create an unprotected version, assuming a suitable digital ID is available in the Windows certificate store *My*. Since the *store* option defaults to *My* and PLOP automatically locates a suitable ID, the *digitalid* option can be omitted. The *password* option can also be skipped since the private key is protected by the Windows login:

```
plop --inputopt "engine=mscapi" --outfile decrypted.pdf encrypted.pdf
```

7 Digital Signatures with PLOP DS

Note *The ability to digitally sign PDF documents is only available in PDFlib PLOP DS, but not in the PLOP base product.*

7.1 Introduction

7.1.1 Basic Concepts of Digital Signatures

Explaining the details of digital signatures is beyond the scope of this manual. However, we will mention some important components which play a role for digitally signing PDF documents with PLOP DS. These components collectively form a Public Key Infrastructure (PKI).

Digital signatures are based on Public Key Cryptography, also called asymmetric encryption. It works with a private key which is only available to the person who signs a document, and a public key which is available to everyone so that they can validate the signatures.

Certificates. Public keys are generally distributed in a so-called certificate which contains the signer's public key and his name and contact details. In order to avoid forged certificates this information package is again signed by a trusted third party which issues a certificate to a person or other entity, such as an enterprise or a server. Such trusted third parties are called Certificate Authority (CA) or Trust Center (TC). The CA's own certificate is called the root certificate. It is usually published on the CA's web site for everyone to download it. Certificates are generally stored in X.509 format.

Since encrypting a document with certificate security requires only the public key, the recipient's certificate is sufficient. On the other hand, decrypting such a document requires the private key which is only available in the recipient's digital ID (see below).

Certificate chain. A signing certificate issued by a CA is considered trustworthy if the issuing CA or the higher-up CA which issued the intermediate CA's certificate is considered trustworthy. The list of certificates which are linked by signing the respective next certificate from the root CA down to the end-user certificate which is actually used to sign a document is referred to as the certificate chain. The top-level CA certificate in the chain is called the root certificate. In order for a signature to be regarded as valid all certificates in the chain must be valid.

Digital IDs. It is important to distinguish certificates from a package containing both the certificate and the corresponding private key, which is called a digital ID. While certificates can freely be distributed to everyone, digital IDs must be carefully protected since they contain confidential information (the private key). Accessing the private key in a digital ID in order to apply a digital signature or to decrypt a document protected with certificate security usually requires a password or passphrase. A common storage format for digital IDs is PKCS#12 (also called PFX on Windows). Note that certificates and digital IDs are not always clearly distinguished: it is common to talk about *signing a document with a certificate* when it would be more accurate to call it *signing with a digital ID*.

Certificate revocation checking. Certificates are valid for a certain period of time. They are no longer valid as soon as their expiration date has passed, or if they have explicitly been revoked by the CA. Revoking a certificate may be necessary because the certificate holder has left the associated organization or the private key has been compromised.

Certificate checking usually involves an online query using a protocol called OCSP (Online Certificate Status Protocol) or certificate revocation lists (CRLs). More details about both methods can be found in »OCSP overview«, page 112, and »CRL overview«, page 114.

Timestamping. Timestamps apply a digital signatures to the representation of a particular point in time, where the time may be obtained from a trusted and accurate time source. Timestamps can be integrated into a regular signature to ensure that the signature and the signed document existed before a certain point in time. Timestamps can also be applied separately to PDF documents. See Section 7.5.1, »Timestamp Configuration«, page 118, for more information about timestamping servers and protocol details.

Sources of digital IDs. There are various sources where you can obtain a digital ID. Many IDs are intended for signing e-mail; these e-mail IDs can also be used in PLOP DS for signing PDF documents. Your choice of source for a digital ID depends on the number of required IDs (e.g. one per employee or only one corporate ID) and the desired degree of control:

- Obtain a digital ID from one of the public CAs which issue commercial or free IDs. In order to facilitate signature validation with Acrobat it is recommended to create signatures with a digital ID from a CA which is installed as trusted root in Acrobat (see »Trusted Root Certificates in Acrobat«, page 90).
- For large organizations: Build your own private CA so that you can create digital IDs yourself. There are various software packages available for building a CA.
- For testing purposes or exchange within a controlled or small user group: Create a digital ID from a self-signed certificate. You can create self-signed certificates in Acrobat DC as follows: *Edit, Preferences, Signatures, Identities & Trusted Certificates, More..., Add ID, A new digital ID I want to create now*
  In the next step you can specify a PKCS#12 disk file or the Windows certificate store as target. Both methods are supported in PLOP DS.

7.1.2 Signatures in Acrobat and PDF

PDF supports different kinds of digital signatures which are discussed below. Signatures are implemented as form fields in PDF. PDF signatures always relate to the whole document (as opposed to single pages) and are available in two flavors:

- Invisible signatures do not occupy any space on the page. They can be viewed in Acrobat by bringing up the *Signatures* pane (Acrobat DC: *View, Show/Hide..., Navigation Panes, Signatures...*).
- Visible signatures use a rectangular form field which is located somewhere on a page in the document. You can specify the page number, field name and field coordinates.

Additional properties can be specified for both types of signatures, e.g. location, reason for signing, and contact information.

Approval signatures. The most common signature type used for PDF is called signature. A PDF document may contain one or more approval signatures. An approval signature is placed in a form field of type *signature* which may be visible or invisible. An approval signature ensures that the document has been signed by the holder of the digital ID and also makes sure that document changes can be detected. Any change applied to the document invalidates the signature. Approval signatures are related to an individual person or entity who creates the signature. Since nobody else has access to the necessary credentials the signer cannot deny the state of the document at signature time (non-repudiation).

When opening a document with an approval signature Acrobat usually displays a blue document message bar near the top of the window (unless the document conforms to PDF/A or contains form fields, in which case the PDF/A status or field message has priority over the signature information). If the signature is valid the message bar contains a green check mark. The signature is also shown in Acrobat's *Signatures* pane.

Approval signatures may optionally contain certificate revocation information and a timestamp for long-term validation. Both items are obtained from a trusted server over the network when the signature is created.

Approval signatures are the default signature type in PLOP DS. They require at least PDF 1.6 output. If necessary, PLOP DS increases the PDF version accordingly.

Approval signatures are reported in pCOS as *signaturefields[...]/sigtype=approval*.

Certification signatures. The first signature in a document may be a certification signature. This type is also called author signature because it certifies the state of the document as the author created it. The document author may allow certain types of modifications which can be applied to the document without breaking the signature. Certification signatures are therefore also called *Modification Detection and Prevention* (MDP) signatures. The following types of allowed modifications can be specified (see Table 7.6):

- No changes allowed: useful for typical read-only documents such as press releases, government publications, etc. In this case even adding an approval or document-level timestamp signature invalidates the certification signature.
- Form filling and adding digital signatures (by clicking a signature field, but not via Acrobat's menu items) allowed: the certification signature ensures form users that they are working with the authentic document, e.g. a purchase order form. When they fill in editable form fields or apply an approval signature the certification signature is not invalidated. Adding pages by spawning page templates is also allowed (as opposed to manually adding pages), but this technique is rarely used.
- Form filling, adding digital signatures and annotations allowed: this could be used e.g. by a notary who wishes to add a comment to a signed document where the comment contains details about the nature of the attestation.

When opening a document with a certification signature Acrobat displays a badge in the document message bar near the top of the window. The signature is also shown in Acrobat's *Signatures* pane (again with a badge if it is valid).

Certification signatures can be created with PLOP DS with the *certification* signature option (see Section 7.3.6, »Certification Signatures«, page 109). They require at least PDF 1.6 output. If necessary, PLOP DS increases the PDF version accordingly.

Certification signatures are reported in pCOS as *signaturefields[...]/sigtype=certification*.

Document-level timestamp signatures. Timestamp signatures must not be confused with an embedded timestamp in an approval or certification signature. A document may contain any number of timestamp signatures. A timestamp signature ensures that the document existed at a particular point in time. The timestamp is obtained from a trusted server over the network and is not related to an individual person or entity who signed the document. Timestamp signatures play an important role for long-term validation since they can be used to refresh existing signatures. Timestamp signatures are placed in a form field, but they are always invisible.

When opening a document with a timestamp signature Acrobat displays a green check mark in the document message bar near the top. The signature is also shown in the *Signatures* pane (with a clock-and-stamp icon if it is valid).

Timestamp signatures can be created with PLOP DS with the *doctimestamp* signature option (see Section 7.5.3, »Document-Level Timestamp Signatures«, page 120). They require at least PDF 1.7ext8 output. If necessary, PLOP DS increases the PDF version accordingly.

Timestamp signatures are reported in pCOS as *signaturefields[...]/sigtype=doctimestamp*.

Usage rights signatures. A document may contain up to two usage rights signatures. They can be used to enable certain editing features in Adobe Reader, resulting in so-called Reader-enabled PDF documents. Usage rights signatures are not bound to signature form fields and are not shown in Acrobat's *Signatures* pane.

Usage rights signatures cannot be created with PLOP DS, but they can queried with the pCOS pseudo object *usagerights*.

7.1.3 Trusted Root Certificates in Acrobat

Adobe Reader and Acrobat accept CA certificates from the sources listed below. These are called trusted roots or trust anchors. You can display Acrobat's list of trusted root certificates with *Edit, Preferences, Signatures, Identities & Trusted Certificates, More..., Trusted Certificates* (see Figure 7.1). Certificates which chain to one of the certificates in the trusted root list certificates are considered trustworthy. Since no Acrobat configuration by the end user is required in order to successfully validate signatures which chain to one of the certificates in Acrobat's root store, it is recommended to create signatures with certificates under the AATL or EUTL root CAs described below.

Adobe Approved Trust List (AATL). The AATL[1] contains commercial, institutional and governmental certificate authorities (CAs) from many countries around the world. At the time of writing dozens of CAs participate in the AATL program.

AATL root certificates are built into Acrobat and Adobe Reader. Acrobat trusts these root certificates and all certificates which chain up to one of these trusted roots. No manual configuration is required to establish this trust relationship. The list can be updated automatically on a regular basis, or manually via *Edit, Preferences, Trust Manager, Automatic Adobe Approved Trusted Certificates Updates*.

AATL CAs issue certificates only on a secure token certified according to FIPS 140-2 Level 2, certified as a Secure Signature Creation Device (SSCD) according to EU regula-

1. See helpx.adobe.com/acrobat/kb/approved-trust-list1.html for more information and a list of participating CAs.

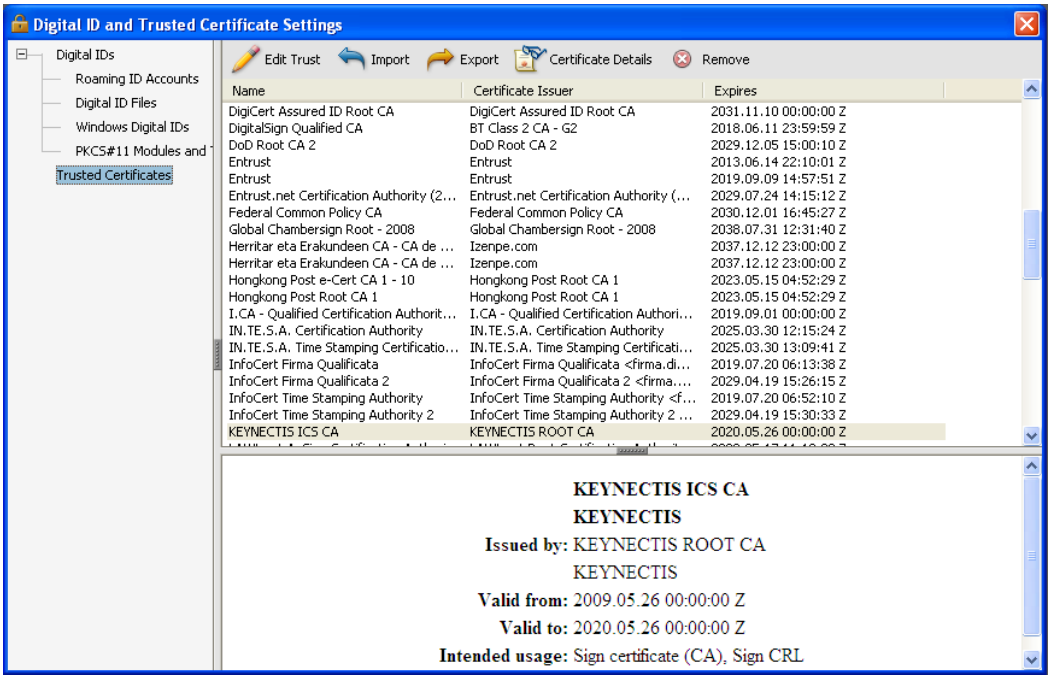


Fig. 7.1
List of trusted certificates in Acrobat

tions, or equivalent standards. In many cases the certificate is stored on a SafeNet token or a Hardware Security Module (HSM). AATL certificates are never distributed on file, but only on a secure token.

Some CAs issue both AATL and non-AATL certificates under the same root. In this case the certificate policy must explicitly state that the certificate has been issued in compliance with the AATL rules. Otherwise Acrobat does not consider it as valid under the known trusted root CAs.

Acrobat also includes CA certificates from Adobe's older Certified Document Services (CDS) program which was introduced in 2005 and is the predecessor of AATL. While AATL CAs are directly treated as a trusted root in Acrobat, CDS certificates chain to the Adobe Root certificate. The following CAs are part of the CDS program: Entrust, GlobalSign, , and DigiCert.

European Union Trust List (EUTL). Adobe Reader and Acrobat DC supports trusted root certificates from the European Union Trust List (EUTL) according to ETSI TS 119 612[1]. The EUTL update can be controlled via *Edit, Preferences, Trust Manager, Automatic European Union Approved Trusted Certificates Updates*. The EUTL includes root certificates from the Trusted Lists of all EU member states according to the eIDAS framework according to regulation 910/2014.

Manually adding trusted roots. Acrobat also accepts root certificates imported manually into Acrobat or Adobe Reader via *Edit, Preferences, Signatures, Identities & Trusted Certificates, More..., Trusted Certificates*. The certificate must be configured as trusted root

1. See digital-strategy.ec.europa.eu/en/policies/eu-trusted-lists

via *Edit Trust, Trust* tab, and activating *Use this certificate as trusted root*. This may be useful for enterprise PKIs with a custom root CA. While this configuration works for any trusted root certificate, it requires manual intervention by the user and is therefore undesirable in some workflows.

Certificates in the Windows certificate store. Acrobat optionally treats root certificates in the Windows certificate store as trusted. This can be controlled via *Edit, Preferences, Signatures, Verification, More..., Windows Integration*.

7.2 Signing with PLOP DS

7.2.1 Overview

PLOP DS supports multiple cryptographic engines which implement the public key and hashing algorithms required for digitally signing a document. Signatures are prepared in the PLOP DS library with *prepare_signature( )* and *create_document( )* API method or the option *--signopt* (shorthand notation: *-S)* of the PLOP DS command-line tool.

In order to apply a digital signature with PLOP DS you need a digital ID. If you work with a digital ID file or token you need the corresponding password. If you work with a personal (account-specific) digital ID in the Windows certificate store the ID is usually protected by your Windows login.

Crypto engines for creating digital signatures. PLOP DS supports various cryptographic engines. A cryptographic engine is a piece of software or hardware which implements various cryptographic functions that are required to generate digital signatures. The choice of a cryptographic engine affects the format and storage location of digital IDs, integration with other software and the operating system. PLOP DS supports the following cryptographic engines:

- The *builtin* engine implements the required cryptographic functions directly in the PLOP DS kernel, without any external dependencies. This engine is active by default, but can also be selected explicitly with the signature option *engine=builtin*.
- The *pkcs#11* engine refers to a software interface called PKCS#11 which provides unified access to cryptographic tokens, where token stands for a smartcard, USB stick or other cryptographic device. Tokens offer higher security than software certificates, and are often protected with a PIN. The PKCS#11 engine is also used for accessing a Hardware Security Module (HSM). The *PKCS#11* engine can be selected with the signature option *engine=pkcs#11*.
- The *mscapi* engine refers to the Microsoft Cryptographic API (available only on Windows), which is an integrated part of the operating system. It allows PLOP DS to interoperate with the cryptographic infrastructure provided by Windows as well as third-party software or hardware which is attached via a CAPI driver. The *mscapi* engine can be selected with the signature option *engine=mscapi*.
- Alternatively a user-supplied cryptographic engine can be used to ensure that all cryptographic operations (hashing and signing) are performed in a dedicated cryptographic library. Attaching such an external cryptographic module requires a special PLOP build which is available on request.

Supported formats for digital IDs. PLOP DS requires a digital ID for signing PDF documents. A digital ID contains the signer's digital certificate plus the corresponding private key, and is usually protected by a password or other means. PLOP DS supports the following kinds of digital IDs:

- With *engine=builtin:* digital ID files in PKCS#12 format (usually *.p12 or.pfx)*
- With *engine=pkcs#11:* digital IDs stored on a smartcard or other cryptographic token (device) attached to the computer.
- Windows with *engine=mscapi:* digital IDs in the Windows certificate store.

7.2.2 Signing with the built-in Engine

The built-in engine is the default engine. It works with file-based digital IDs and provides full functionality and control.

Unlocking the private key. Digital IDs (more precisely: the private key contained in the digital ID) are generally protected with a password, passphrase, or PIN since they contain the confidential private key for creating the digital signature. In order to unlock a digital ID for use with PLOP DS you must provide proper authentication. If you supply the wrong password PLOP DS will throw an exception.

You must supply the corresponding password with the *password* signature option. If you are using the PLOP DS command-line tool it is strongly recommended to supply the password indirectly in an auxiliary file with the *passwordfile* suboption. If you supply the password directly instead of in a password file other users could possibly read it since the command-line may be visible to other users on a multi-user system.

Option list example. The examples below show how to digitally sign PDF documents with the PLOP DS command-line tool. The option list supplied to *--signopt* can be supplied to the PLOP DS API method *prepare_signature( )* in order to create a signature from within your own program. Full programming examples for all supported language bindings are contained in the PLOP DS package. The examples use the digital ID file *demo_signer_rsa_2048.p12* with the password *demo* which is included in the distribution packages.

Create an invisible signature for a PDF document, using a digital ID from the file *demo_signer_rsa_2048.p12*. The password for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt" ↵
       --outfile signed.pdf input.pdf
```

7.2.3 PKCS#11 Engine for a cryptographic Token

Using the PKCS#11 engine in PLOP DS you can use certificates on a cryptographic token such as a smartcard or USB stick, or on a Hardware Security Module (HSM). Using such a device for signature creation requires a DLL or shared library which implements a token-specific protocol. This PKCS#11 DLL/SO is provided by the token vendor as part of the corresponding software package. It must be installed on the system and made available to PLOP DS. On Windows this means the DLL must either be copied to the Windows system directory, a directory which is included in the PATH environment variable, or the current directory of the application. Note that a PKCS#11 DLL/SO may depend on other DLLs. In this case all required DLLs supplied by the vendor must be made available to PLOP DS.

Selecting a private key. A signature device may contain multiple digital IDs, e.g. one for encrypting E-mails and another one for digitally signing documents. If exactly one signature certificate is present on the token PLOP DS automatically selects it. If multiple signature certificates are present on the token you must supply one of the suboptions *issuer*, *label*, *serial* or *subject* of the *digitalid* option to select the appropriate certificate by one of these criteria. If the supplied options do not select exactly one signature certificate the call fails and no signatures can be created. A label can be assigned to a key with

Fig. 7.2
Smartcard reader with keyboard (left) and cryptographic USB token (right). Both devices can be attached to PLOP DS with the PKCS#11 engine.

the administration software for the token. Issuer, serial number and subject are intrinsic fields of the certificate.

Unlocking the private key on a token. If the cryptographic token allows a password or PIN to be submitted by software you must supply the *password* signature option as with *engine=builtin*. If the token requires direct PIN or password entry (e.g. a smartcard reader with attached keyboard) you can omit the *password* option (or supply an empty string) and must manually type the PIN on the token's keyboard. Details of password/PIN handling vary among cryptographic tokens.

Some tokens automatic log out after a certain period of time or a specified number of signatures. For mass signatures you must configure the token appropriately to avoid automatic logout. Please refer to your token documentation for details. If the token automatically logs out you will experience the following error message:

```
Error adding signature data ('PKCS#11: couldn't create signature
(C_Sign: CKR_USER_NOT_LOGGED_IN)')
```

PKCS#11 example. In the following examples we refer to the vendor-specific PKCS#11 DLL as *cryptoki.dll*. The name of the actual DLL may be different.

Create an invisible signature for a PDF document, using a digital ID from a token addressed via PKCS#11. The PIN for the token is contained in the file *pw.txt:*

```
plop --signopt "engine=pkcs#11 digitalid={filename=cryptoki.dll} passwordfile=pw.txt" ↵
        --outfile signed.pdf input.pdf
```

Create an invisible signature for a PDF document, using a digital ID from a token addressed via PKCS#11. No PIN is supplied in this command; instead, the PIN for the token must be typed at the token's integrated keyboard:

```
plop --signopt "engine=pkcs#11 digitalid={filename=cryptoki.dll}" ↵
        --outfile signed.pdf input.pdf
```

7.2.4 PKCS#11 Engine for a Hardware Security Module (HSM)

A Hardware Security Module (HSM) provides hardware-based security for the private key and massive performance advantages compared to tokens or smartcards. HSMs are typically deployed in the following scenarios:

- Commercial CAs offer HSM-based certificates. The HSM is typically run and managed by the CA. For demanding applications the HSM may reside on the customer premises. For example, GlobalSign, QuoVadis and Symantec offer HSM-based AATL certificates.
- In-house HSM deployment with an enterprise PKI.
- Cloud-based signatures: HSM services can be purchased along with CPU and storage services. For example, IBM, Amazon Web Services (AWS) and Microsoft Azure offer HSM hosting.

In our testing we found that the full HSM performance can be obtained only with a multi-threaded client application or many independent simultaneous clients.

Supplying the signer's certificate. The HSM creates a private/public key pair and safely stores the private key on the device. However, some HSMs don't offer any straightforward way (or none at all) of storing the signer's certificate with the corresponding public key. Since the certificate must be embedded in the PDF signature you must make the signer's certificate available on file so that PLOP DS can embed it in the signed PDF. This can be achieved with the *signercert* suboption of the *digitalid* option. This option is required for HSMs which hold only the private key, but not the corresponding public key. The user is responsible for making sure that the certificate supplied with the *signercert* option matches the private key selected with one of the *id* or *label* options. If they don't match, an invalid signature will be created.

PKCS#11 example for nCipher nShield HSM (formerly Thales nShield HSM)**.** The following example demonstrates digital signatures with a nCipher nShield HSM. We assume that a key pair has been generated with the HSM, a corresponding certificate is available in a separate file and a suitable security world has been configured with the administration software that accompanies the nShield appliance (refer to nShield documentation for details). In this situation PDF documents can be signed as follows:

```
plop --signopt "engine=pkcs#11 digitalid={label=demo_signer_rsa_2048 ↵
        filename={/opt/nfast/toolkits/pkcs11/libcknfast.so} ↵
        signercert={demo_signer_rsa2048.crt}}" -o signed.pdf input.pdf
```

Using a somewhat convoluted procedure the signer's certificate can be stored directly on the HSM device. In this situation PDF documents can be signed with the following alternative command:

```
plop --signopt "engine=pkcs#11 digitalid={label=demo_signer_rsa_2048 ↵
        filename={/opt/nfast/toolkits/pkcs11/libcknfast.so}}" -o signed.pdf input.pdf
```

Note *Please also take a look at the separate document »nCipher e-Security PDFlib PLOP DS Integration Guide«.*

PKCS#11 example for AWS CloudHSM. AWS CloudHSM v2 stores only the private key on the device, but does not support certificate storage. Since the certificate must be em-

Fig. 7.3
A Hardware Security Module (HSM) can be attached to PLOP DS with the PKCS#11 engine

bedded in the PDF signature you must make the signer's certificate available on file. This can be achieved with the *signercert* suboption of the *digitalid* option:

```
plop --signopt "password={<username>:<pin>} engine=pkcs#11 ↵
        digitalid={filename={/opt/cloudhsm/lib/libcloudhsm_pkcs11_standard.so} ↵
        label=demo_signer_rsa_2048 signercert={demo_signer_rsa2048.crt}}" ↵
        -o signed.pdf input.pdf
```

When testing an application with multiple PLOP objects on CloudHSM we found that *prepare_signature( )* sometimes fails with the following error message:

```
Error in PKCS#11 operation ('couldn't open session (C_OpenSession: CKR_DEVICE_MEMORY)')
```

and corresponding console output created by the PKCS#11 library:

```
C_OpenSession failed with error CKR_DEVICE_MEMORY : 0x00000031
```

when attempting to load the PKCS#11 library again after a previous PLOP object has already been deleted. This error seems to be caused by a problem in the PKCS#11 implementation of CloudHSM. The problem can be avoided by supplying the *sticky* suboption of the *digitalid* option.

PKCS#11 sessions and multi-threading. In order to improve performance of bulk signatures, PLOP DS minimizes the number of load/unload operations for the PKCS#11 DLL/SO, and maximizes the duration of each PKCS#11 session. This requires the application to obey to the following conditions:

- At any time only a single PKCS#11 DLL/SO can be loaded until *delete( )* has been called for the last PLOP object which used that library. After deleting the last PLOP object another PKCS#11 DLL/SO can be specified in *prepare_signature( )*. In other words, any number of PKCS#11 slots can be addressed in a multi-threaded manner provided all token slots are served by the same DLL/SO (which usually means the same type of token).
- *prepare_signature( )* in a particular thread must not access a PKCS#11 slot which is already accessed from another thread. Multi-threaded applications which want to sign with the same token from within multiple threads must synchronize the threads by suitable means, e.g. a mutex.
- Multi-threaded applications require a thread-safe PKCS#11 library. The suboption *threadsafe=true* in the *digitalid* option list checks whether the library is thread-safe and initializes it in a thread-safe manner.
- The PKCS#11 DLL/SO can optionally be kept in memory even after the last use with the *sticky* suboption of the *digitalid* signature option. This may slightly speed up signature operation. However, no other PKCS#11 DLL/shared library can be loaded in the same process.

A new session is created in the first call to *prepare_signature( )* for a particular slot and maintained until *prepare_signature( )* is called again in the same thread. If no more signatures will be created in a thread, the PKCS#11 session can explicitly be terminated by calling *prepare_signature( )* with the option *signature=false*. For this reason the application should call *prepare_signature( )* only once for as many output documents as possible. For example, as long as the same PKCS#11 slot is addressed and the token's restrictions are met (e.g. maximum number of signatures or maximum time for consecutive signatures) no more calls to *prepare_signature( )* are required.

A full code sample for efficiently applying bulk signatures is available in the *multisign* sample which is part of all PLOP DS packages. Note that the *multisign* logic offers significant performance advantages for token-based signatures in comparison to the PLOP DS command-line tool.

7.2.5 Signing with the MSCAPI Engine on Windows

Using the MSCAPI engine allows you to take advantage of the signature features built into the Windows operating system. Most importantly, you can access digital IDs in the Windows certificate store. On the other hand, the MSCAPI engine is subject to certain restrictions which don't affect other cryptographic engines. For example, MSCAPI does not support ECDSA.

Note OSCP and CRL embedding as well as timestamping are not supported for `engine=mscapi`. *As a result, LTV-enabled signatures can not be created with the MSCAPI engine.*

Unlocking the private key. Depending on your certificate settings the digital IDs in the Windows certificate store may be protected by your Windows login, and no additional password is required. If you enabled high security when importing the certificate into the Windows certificate store you are prompted for the password whenever the certificate is used for signing.

Option list examples for MSCAPI. The examples below assume that the digital ID for signing is available in the Windows certificate store. In order to achieve this with the PLOP DS demo certificates you must double-click and install the digital ID in the file *demo_signer_rsa_2048.p12* in the Windows certificate store.

Create an invisible signature for a PDF document, using a certificate from the Windows Certificate Store (from the default store *My* and the default store location *current_user)*. This assumes that the digital ID is protected by your Windows login so that no password must be supplied:

```
plop --signopt "engine=mscapi digitalid={store=My subject={PLOP Demo Signer RSA-2048}}" ↵
         --outfile signed.pdf input.pdf
```

Create an invisible signature for a PDF document, using a certificate in the file *demo_signer_rsa_2048.p12:*

```
plop --signopt "engine=mscapi digitalid={filename=demo_signer_rsa_2048.p12} ↵
        passwordfile=pw.txt"  --outfile signed.pdf input.pdf
```

Create an invisible signature and encrypt the document with the master password *SECRET* for PDF encryption and password *demo* for accessing the digital ID:

```
plop --master SECRET --signopt "digitalid={filename=demo_signer_rsa_2048.p12} ↵
        password={demo}" --outfile signed.pdf input.pdf
```

Managing the Windows certificate store. The Windows operating system can hold certificates which are organized in several certificate stores. To install a new certificate in PKCS#12 format simply double-click on the certificate file and follow the Certificate Import Wizard. You can try this with the demo certificates in the PLOP DS package, using the password *demo*.

You can view and organize certificates with the Microsoft Management Console (MMC) as follows:

- Click on *Start* and type *mmc* in the box for program names to launch the program.
- In the *File* menu click *Add/Remove Snap-in...*
- In *Available Standalone Snap-ins* select *Certificates* and click *Add*.
- In the next dialog select *My user account* and *Finish*. Alternatively, use *Service account* or *Computer account* if this is the store location of your certificates.
- Click *OK*.

Now you can browse the installed certificates. Your own certificates are available in the *Personal* category, which can be addressed in PLOP DS with the following option list (supplied to the *--signopt* command-line option or *prepare_ signature( ))*:

```
engine=mscapi digitalid={store=My subject={PLOP Demo Signer RSA-2048}}
```

You can view certificate details by double-clicking on a certificate in MMC. In order to export a certificate in PFX format right-click on a certificate in the list and click *All Tasks, Export...* . This launches the Certificate Export Wizard.

Using the Management Console you can also import a certificate: right-click on a certificate store (e.g. *Personal)* and select *All Tasks, Import...* .

7.2.6 Cryptographic Details

Digital signatures are characterized by an encryption algorithm and a hash algorithm plus parameters for both. Encryption algorithm and key length for generating signatures are determined by the signer's digital ID. They are specified when creating the public/private key pair for the digital ID. PLOP DS supports the signature algorithms listed below.

RSA signatures. RSA is supported with key lengths in the range 1024-8192 (3072-bit or more recommended). RSA is widely used on the Internet and many other application areas. RSA signatures require a so-called encoding method *(Encoding Method for Signatures with Appendix, EMSA):*

- The default encoding method according to PKCS#1 v1.5 is supported in all Acrobat versions. However, it is being phased out in many signature applications.
- The newer EMSA-PSS *(Probabilistic Signature Scheme)* encoding method according to RFC 3447/RFC 8017 offers provable security. EMSA-PSS is also called SSA-PSS or PKCS #1 v2.1. EMSA-PSS signatures can be created with the signature option *rsaencoding=pss*.

  Acrobat DC displays the RSA encoding method in the *Advanced Signature Properties* dialog (see Figure 7.4).

Note EMSA-PSS signatures are not supported for `engine=mscapi`.

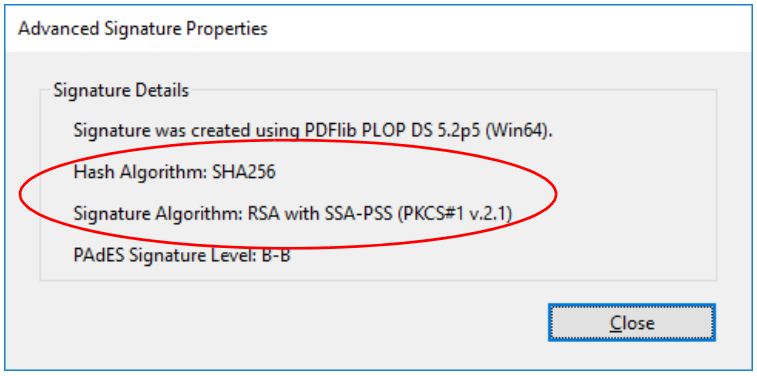


Fig. 7.4
Acrobat displays the hash algorithm and encoding method of a signature

DSA signatures. DSA is supported with key lengths in the range 1024-4096 (3072-bit or more recommended). DSA is not widely used. Since Acrobat supports only DSA with the insecure SHA-1 hash algorithm, there are security concerns regarding the use of DSA.

Note DSA signatures are not supported for `engine=pkcs#11`.

Elliptic curve cryptography. ECDSA (Elliptic Curve Digital Signature Algorithm) is the modern successor of RSA. Key lengths commonly go up to 512-bit (256-bit or more recommended). The main advantage of ECDSA over RSA is a smaller key size to achieve the same cryptographic strength, which in turns implies performance benefits. The strength of ECDSA is determined by a curve which is characterized by parameters or more commonly a name. There are three common groups of ECDSA curves:

- The most common curves have been standardized by NIST and listed in RFC 5480. They are called *P-256, P-384,* and *P-521*; other names are *secp256r1* (or *prime256v1*), *secp384r1,* and *secp521r1*. These curves are supported in Acrobat DC.
- RFC 5480 defines an additional set of 12 named curves recommended by NIST. These are also supported in Acrobat DC when loading the digital ID directly in Acrobat.
- RFC 5639 defines a set of curves called Brainpool curves. Signatures based on Brainpool curves cannot be validated with Acrobat DC. Unfortunately, Acrobat doesn't clearly indicate that the signature algorithm is unsupported, but instead issues the following error message for Brainpool signatures:

  ```
  There are errors in the formatting or information contained in this signature.
  ```

  Since Acrobat DC cannot validate signatures based on Brainpool curves, these require the signature option *conformance=extended*.

Note ECDSA signatures are not supported for `engine=mscapi`.

Hash Algorithms. A hash algorithm is used to create a message digest for the signed data. Common hash algorithms are SHA-1 (no longer considered secure) and the stronger algorithms in the SHA-2 family which includes SHA-256, SHA-384 and SHA-512. The hash algorithm used for a signature can be displayed in Acrobat DC as follows (see Figure 7.4):

- open the *Signatures* pane;

- select a signature and select *Show Signature Properties...* in the *Signatures* menu.
- click *Advanced Properties... ;*
- the resulting dialog entitled *Advanced Signature Properties* displays *Signature Details* including hash algorithm and encoding method.

Table 7.1 lists signature algorithms and corresponding hash functions as well as the minimum PDF output version created for each signature algorithm. If the input document uses a lower PDF version number PLOP DS increases the PDF version of the output document to the one listed in the table.

Table 7.1 Signature algorithms, hash algorithms, PDF output version and required Acrobat versions

signature algorithm	***hash algorithm***	***PDF output version[1]***
Approval and certification signatures		
RSA up to 8192 bit	*SHA-256*	`sigtype=cades`: *PDF 1.7ext8* `sigtype=cms`: *PDF 1.6*
DSA up to 4096 bit	*SHA-1 (Acrobat DC doesn't support other hash algorithms for DSA)*	`sigtype=cades`: *PDF 1.7ext8* `sigtype=cms`: *PDF 1.6*
ECDSA with NIST curves (RFC 5480) P-256/P-384/P-521	*SHA-256, SHA-384 or SHA-512 depending on the curve*	*PDF 1.7ext8*
ECDSA with NIST curves (RFC 5480) other than P-256/P-384/P-521	*SHA-256, SHA-384 or SHA-512 depending on the curve*	*PDF 1.7ext8*
ECDSA with 14 Brainpool curves (RFC 5639)	*SHA-256, SHA-384 or SHA-512 depending on the curve*	*PDF 1.7ext8 / cannot be validated with Acrobat XI/DC* *(requires* `conformance=extended`)
Document-level timestamps		
determined by the TSA	*SHA-256 by default, but can be changed with* `doctimestamp` *sub-option* `hash`	*PDF 1.7ext8 with PAdES part 4 extension*
OCSP request and response (certificate identification)		
determined by the OCSP responder	*SHA-1 by default, but can be changed with* `ocsp` *suboption* `hash`	*Acrobat DC and below support only SHA-1 for OCSP;* *(other hash functions require* `conformance=extended`)

1. In PDF/A and PDF/X modes the PDF version of the input document remains unchanged.

7.3 PDF Aspects of Signatures

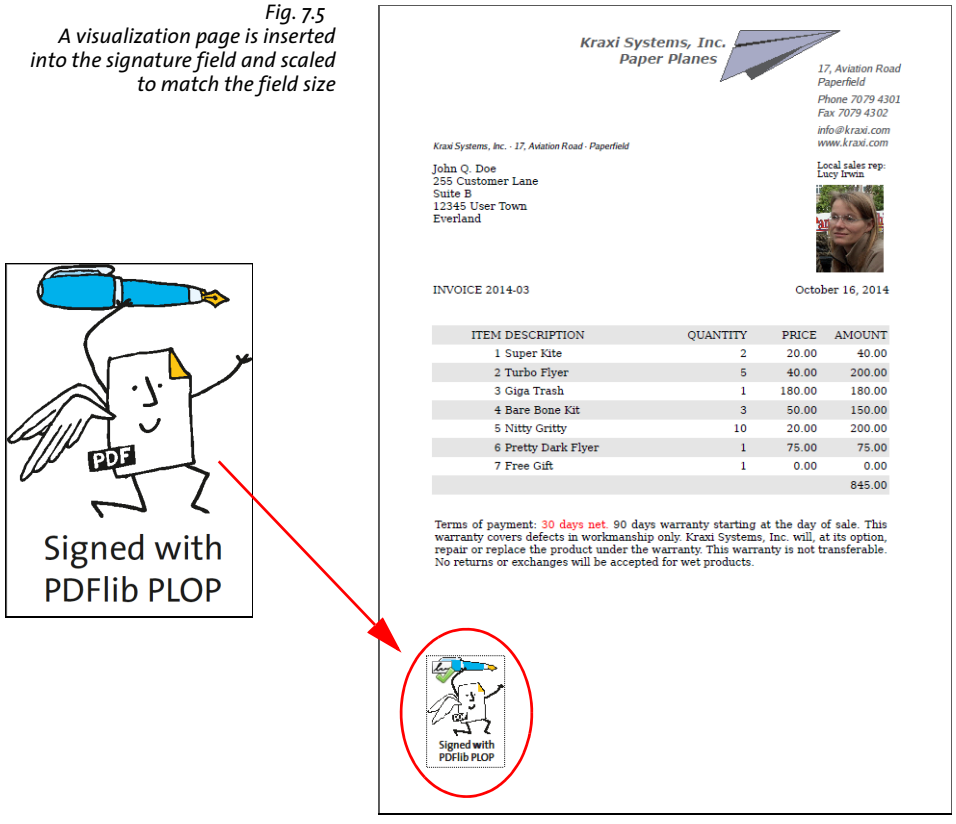
7.3.1 Visualizing Signatures with a Graphic or Logo

Digital signatures can be integrated in the document in the following ways:

- Invisible signatures don't have any representation on a page. They are shown in Acrobat's *Signatures* pane only. Document-level timestamp signatures are always created as invisible signatures.
- Visible signatures may contain arbitrary text or graphics to display the signature visually at a specific location on a page (see Figure 7.5). A page from an existing PDF document can be used to create the signature's visual appearance. Visual signatures are also represented in the *Signatures* pane. The document from which this page is taken is called the visualization document. Since the visualization page is placed in the field which holds the signature, you can click the visualization in the signed document to validate the signature in Acrobat.

Note *Although technically it would be possible to repeat the signature visualization on multiple pages, this is not supported in PLOP DS because of legal uncertainties related to a non-unique signature visualization. Because of these uncertainties repeated signature visualizations are explicitly forbidden in PDF 2.0.*

Fig. 7.5
A visualization page is inserted into the signature field and scaled to match the field size



Signature visualization document. The PDF page used for signature visualization may contain a scanned hand-written signature, an official seal or a company logo, a photo of the holder of the signing certificate, or any other visible representation which may be useful to recipients of the signed document.

If the visualization document uses a higher PDF version than the signed input document, the PDF version of the generated output is adjusted accordingly. *PDF 1.7ext3* (Acrobat 9) and *PDF 1.7ext8* (Acrobat X/XI/DC) documents are compatible with PDF 1.7 regarding their use as visualization page.

Note *Signature visualization for PDF/A imposes certain conditions on the visualization document (see »PDF/A conformance«, page 104). Visualizing digital signatures is not supported in PDF/X and PDF/VT modes.*

The visualization document must be opened with *open_document( )*. You must supply its document handle to the *visdoc* suboption of the *field* option:

```
field={visdoc=<handle> rect={100 100 300 150}}
```

Location and size of the signature field. The *field* signature option controls the representation of the signature on the page. Location and size of the signature visualization page on the visible page of the signed document can be specified with the *rect* suboption of the *field* option. The size can be specified explicitly, or implicitly by specifying one corner and one or two of the other dimensions. The missing values are specified with the keyword *adapt* to calculated automatically to avoid distortion. With the *adapt* keyword you can attach the visualization page to any corner of the signature rectangle. The resulting rectangle must not exceed the page. The examples below demonstrate various combinations:

- The simplest approach is to prepare the visualization page in the desired target size. In this case you can simply supply the coordinates of the lower left corner of the field and PLOP DS will use the original page dimensions for the signature visualization:

  ```
  rect={100 100 adapt adapt}
  ```

- Attach to the lower left corner, maintain the width and adapt the height to avoid distortion:

  ```
  rect={100 100 300 adapt}
  ```

- Attach to the lower left corner, adapt the width and maintain the height to avoid distortion:

  ```
  rect={100 100 adapt 200}
  ```

- Force-fit the page into the rectangle, i.e. maintain both width and height of the rectangle. If the page and the rectangle have different width/height ratios the visualization page appears distorted:

  ```
  rect={100 100 300 200}
  ```

In order to calculate a suitable signature field rectangle dynamically depending on the size of the visualization page you can use the pCOS interface to query the page dimensions (keep in mind that pCOS page indexes start at 0):

```
width  = plop.pcos_get_number(visdoc, "pages[" + (vispage-1) + "]/width");
height = plop.pcos_get_number(visdoc, "pages[" + (vispage-1) + "]/height");
```

Signing into an existing form field. If the input document already contains a signature field you can use this field for the signature and its visualization. In order to achieve this you can supply the name of the existing field if you know it:

```
field={name=MyExistingFieldName visdoc=<handle>}
```

If you don't know the field name you can instruct PLOP to use an existing signature field as follows:

```
field={fillexisting visdoc=<handle>}
```

Even if you sign into an existing field you can modify its position and size with the *rect* field option. If you create a signature into an existing field and the field uses a visible rectangle on the page you must supply the *visdoc* option (or make the field invisible with the field option *rect={0 0 0 0}*).

Placing the visualization page inside the signature field. The visualization page is placed in the signature field and scaled such that it entirely fits into the rectangle while preserving its aspect ratio. This is particularly useful if you want to place the signature in an existing form field and the width/height ratio of the field and the visualization page don't match.

The *position* suboption of the *field* option can be used to specify the placement of the visualization page inside the signature field.

By default, the visualization page is centered horizontally and vertically in the field. This can be changed, e.g. to place the visualization page at the lower left corner of the signature field:

```
field={name=MyExistingFieldName visdoc=<handle> position={left bottom} }
```

pCOS. Signature visibility is reported in pCOS as *signaturefields[...]/visible=true*. The information whether or not a signature field already contains a signature can be queried with *signaturefields[...]/sigtype != none*.

7.3.2 PDF/A, PDF/UA, PDF/X and PDF/VT Conformance

Unless mentioned otherwise in this manual, all PLOP operations conform to PDF/A, PDF/UA, PDF/VT and PDF/X regulations which means that standard conformance is maintained by PLOP. However, there are some exceptions to this rule where PLOP operations are prohibited by a particular standard, e.g. encryption in PDF/A. In such cases you must consider your priorities:

- If you must maintain standard conformance the operation will be rejected by PLOP. This is the default behavior.
- If the operation (e.g. encryption) is more important than standard conformance, you can remove the standard identifier with the *sacrifice* option.

Specific notes on the relevant standards are provided below.

PDF/A conformance. The PDF/A standard allows CMS- and CAdES-based signatures. PDF/A-2 and PDF/A-3 recommend to embed a timestamp, revocation information, and

as much of the certificate chain as is available, but this is not a strict requirement and therefore not enforced by PLOP DS.

In PDF/A mode, i.e. if the input conforms to PDF/A and the *sacrifice* option has not been set to *pdfa*, a signature visualization document must be compatible regarding its PDF/A characteristics:

- The PDF/A level of the visualization document must be compatible (see Table 7.2).
- The output intent of the visualization document must be compatible (see Table 7.3).

Tip: a PDF/A-1a visualization document without an output intent (highlighted in red in Table 7.2 and Table 7.3) is compatible with all PDF/A parts, conformance levels, and output intent types. The PLOP DS distribution contains a sample visualization file *signing_man_pdfa1a.pdf* with these characteristics. It can be used as visualization document for testing with all PDF/A flavors. A PDF/A-1b visualization document without an output intent is compatible with the *b* conformance levels of all PDF/A parts.

If you don't care about PDF/A conformance you can remove the standard conformance entry with the following option:

```
sacrifice={pdfa}
```

Table 7.2 Compatible PDF/A levels of the visualization document for various PDF/A input levels

	PDF/A level of the visualization document				
PDF/A level of the input document	**PDF/A-1a:2005**	**PDF/A-1b:2005**	**PDF/A-2a, PDF/A-3a**	**PDF/A-2b, PDF/A-3b**	**PDF/A-2u, PDF/A-3u**
PDF/A-1a:2005	allowed	–	–	–	–
PDF/A-1b:2005	allowed	allowed	–	–	–
PDF/A-2a, PDF/A-3a	allowed	–	allowed	–	–
PDF/A-2b, PDF/A-3b	allowed	allowed	allowed	allowed	allowed
PDF/A-2u, PDF/A-3u	allowed	–	allowed	–	allowed

Table 7.3 PDF/A output intent compatibility of visualization documents (for all PDF/A conformance levels)

	output intent type of visualization document			
output intent type of the input document	**none**	**Grayscale**	**RGB**	**CMYK**
none	allowed	–	–	–
Grayscale ICC profile	allowed	allowed[1]	–	–
RGB ICC profile	allowed	–	allowed[1]	–
CMYK ICC profile	allowed	–	–	allowed[1]

1. The output intent of the visualization document and the output intent of the input document must be identical.

PDF/UA conformance. The PDF/UA requirements for invisible signature form fields have been relaxed per the »Tagged PDF Best Practice Guide« (published in 2019 by the PDF/UA Competence Center of the PDF Association). In particular, invisible signature fields don't have to be included in the structure hierarchy and don't require any special preparations or signature options.

Note The Accessibility check in Acrobat DC does not implement this relaxed rule. It still reports »Tagged annotations - Failed« for an invisible signature form field which is not included in the structure hierarchy.

In order to use visible signature fields you must prepare a suitable form field with alternate text in the input document; creating a new field is not possible. In Acrobat DC this can be achieved as follows for an existing PDF/UA document:

- Click *Tools, Prepare Form*,select the document, Start. In the list of form tools in the toolbar near the top select the Signature form tool.
- Draw a form field rectangle on the page.
- Close the *Prepare Form* window and open the *Tags* pane.
- Click the options button at the top of the *Tags* pane and choose *Find...* .
- In the resulting dialog select *Unmarked Annotations* and click *Find*.
- The signature field just created should now be highlighted. In the *Find Element* dialog click *Tag Element*, choose *Type: Form*, optionally supply a field title, and click *OK*.
- In the *Tags* pane the newly created *Form* structure element should show up at the end of the tag list. Select the tag and move it to a suitable position in the tags hierarchy, corresponding to the position in the structure tree where you want the signature field to be read.
- It is recommended to assign alternate text for the signature field: right-click the *Form* structure element in the hierarchy, select *Properties...,* and enter suitable alternate text for the field.

Assuming the signature field has been assigned the name *Signature1* you can reference it by name in the signature option list as target field for the signature:

```
field={name=Signature1}
```

Alternatively you can instruct PLOP DS to place the signature in the existing field regardless of its name:

```
field={fillexisting}
```

The *tooltip* suboption of the *field* signature option can be used to provide a suitable alternate description of the signature field for use by screen reader software.

If you don't care about PDF/UA conformance you can remove the standard conformance entry with the following option:

```
sacrifice={pdfua}
```

PDF/X and PDF/VT conformance. Signature visualization is not supported in PDF/X and PDF/VT modes.

If you don't care about PDF/X conformance you can remove the standard conformance entry with the following option (similar for PDF/VT):

```
sacrifice={pdfx}
```

7.3.3 Document Security Store (DSS)

A dedicated PDF data structure called Document Security Store (DSS) can hold certificates and related OCSP and CRL revocation information. This material is collectively called validation information and plays an important role for long-term validation. The DSS has been introduced with PAdES part 4 and is included in ISO 32000-2. While the

DSS is optional for approval and certification signatures, it is required for enabling long-term validation of document timestamps and timestamped signatures.

Storing validation information in the DSS instead of in the signature object reduces the file size because unlike the signature object the DSS can be compressed and doesn't require ASCII representation (which doubles the size of the signature). Also, the DSS may hold data for validating multiple document signatures, while the signature object holds only validation information for a single signature.

Some pieces of validation information can be stored only in the signature object, some only in the DSS, and some in both locations. The signature option *dss* can be used to control the storage location of the items in the last group. Table 7.4 compares both locations.

Table 7.4 Storage locations for various pieces of validation information

	signature object	***Document Security Store (DSS)***	***controlled by dss option***
signing certificate	*yes*	–	–
TSA certificate	–	*yes*	–
certificates other than the signing and TSA certificate (e.g. issuer of signing certificate), and corresponding OCSP responses and CRLs	*yes*	*yes*	*yes*
OCSP responses and CRL for the signing certificate	*yes*	*yes*	*yes*
OCSP responses and CRLs for TSA certificates[1]	–	*yes*	–

1. If validation information for timestamps must be embedded, PLOP DS always appends a DSS as incremental update.

PLOP DS preserves an existing DSS with validation information for earlier signatures which may be present in the input document. The new DSS includes the contents of the existing DSS plus validation information for the new signature. This ensures that the LTV status of existing signatures is kept intact.

In Acrobat DC a DSS can be added to a signed document by opening the *Signatures* pane and clicking *Add Verification Information* in the *Options* menu.

pCOS. The presence of a DSS can be checked with the pCOS path *type:/Root/DSS* which has the value 6 *(dict)* if a DSS is present. Note that a DSS in itself does not automatically guarantee LTV status since it could contain only a subset of the required certificates and revocation information.

7.3.4 Signatures and incremental PDF Updates

By default, PLOP DS appends digital signatures to the input document using a PDF technique known as incremental update: a copy of the input document is created and signature data is appended at the end, preserving the contents and structure of the original document. With the signature option *update=false* PLOP DS rewrites the hierarchy of PDF objects instead of adding an incremental PDF update. Table 7.5 compares signatures in update and rewrite mode.

Signing damaged documents. Errors in the PDF cross-reference table or the document's object structure cannot be repaired when signing in update mode. If a document

Table 7.5 Comparison of signing in update and rewrite mode

	Update mode (update=true)	***Rewrite mode*** (update=false)
existing signatures	*preserved*	*lost*[1]
DSS for approval and certification signatures can be added	*yes*	*yes*
DSS for document-level timestamps and timestamped signatures (required for LTV) can be added	*yes*	*–*[2]
existing DSS is preserved	*yes*	*yes*
encryption with new parameters possible (userpassword, masterpassword, permissions)	*–*	*yes*
optimization possible	*–*	*yes*
repair mode for damaged input documents possible	*–*	*yes*
signature speed	*slightly faster*	*slightly slower*
previous document version (before applying the signature) can be recovered in Acrobat	*yes*	*no*

1. Signing documents with existing signatures in rewrite mode requires sacrifice={signatures} *which implies that the signatures are removed. If the* sacrifice *option is not supplied, signed input documents are rejected.*
2. If validation information for timestamps is embedded, PLOP DS always appends a DSS as incremental update.

requires repairing in *open_document( )* and is subsequently signed in update mode, *create_document( )* will fail with the error message

```
Cannot sign damaged input document 'bad.pdf' in update mode; use update=false
(invalid xref table)
```

In order to detect damaged documents already in *open_document( )* you can supply the option *repair=none*. As a result, *open_document( )* will fail for damaged documents. If you need to sign documents which require repair you must use *update=false; see* Table 7.5 for implications.

Reverting to earlier revisions of a signed document. Since incremental updates only add information to a document the structure of the input document is preserved. If a signed document is modified, the signed revision can be reconstructed by removing the incremental updates. In Acrobat DC this can be achieved as follows:

- open the signature pane, select a signature and expand it by clicking the plus sign;
- select *Click to view this version* to revert to the signed revision.

If the signature is LTV-enabled via a DSS in a separate incremental update, this update will be removed by reverting to the signed revision. As a result, the signature in the earlier revision may no longer be displayed as LTV-enabled although the same signature in the full document is displayed as LTV-enabled. This is a result of removing incremental PDF updates and does not affect the actual LTV status of the signatures in the complete document. This issue does not affect timestamp signatures since Acrobat does not require full validation information for the TSA.

This effect only occurs if validation information in a DSS is appended in an incremental update, and can therefore be avoided in two ways:

- set *dss=false* to avoid the DSS;

- set *update=false* to avoid incremental updates.

Both options don't affect document-level timestamps and embedded timestamps which always require a DSS in an incremental update.

pCOS. The number of document revisions by incremental updates is reported in the pCOS pseudo object *revisions*. While each signature creates a new revision, revisions may also be created by other changes, e.g. adding a DSS. The number of revisions therefore may be larger than the number of signatures in the document.

7.3.5 Combining Encryption with Signatures

The combination of encrypting and signing requires attention since first signing and then encrypting a document would invalidate the signature. You can either encrypt and sign in a single pass, or sign an encrypted input document in update mode.

Encrypt and sign in a single pass. The simplest approach is to apply both encryption and signature in a single pass. Since the input file must be modified signing is only possible in rewrite mode. Encryption options, i.e. *userpassword*, *masterpassword* or recipient certificates can be supplied together with signature options. If any of these encryption parameters is supplied, the signature is automatically applied in rewrite mode, i.e. *update* is forced to *false*.

Sign encrypted input documents in update mode. Encrypted input documents can be signed in update mode. However, the encryption parameters cannot be changed in this case. This has the following consequences:

- The input document's master password must be provided in the *password* option (or a suitable digital ID for documents protected with certificate security).
- If *update=true* is supplied, the *encryption*, *masterpassword*, *permissions*, and *userpassword* options are not allowed and *add_recipient( )* must not be called since the values of the input document are used for the output document.

7.3.6 Certification Signatures

Certification (author) signatures have been introduced in »Certification signatures«, page 89. When opening a document with a certification signature Acrobat displays a badge in the blue document message bar near the top and in Acrobat's *Signatures* pane (again with a badge if it is valid). Certification signatures specify which kind of changes may be applied to the document without invalidating the signature (see Figure 7.6 and Table 7.6). Certification signatures can be created with PLOP DS with the *certification* option.

The following signature options create a certification signature such that form filling is allowed without invalidating the signature:

```
digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ↵
certification=formfilling
```

The *preventchanges* suboption can be used to disable tools in the Acrobat user interface which would invalidate the signature, e.g. commenting tools. This way the user is not tempted to apply changes which would break the certification signature. When certifying documents with Acrobat changes are always prevented. The *preventchanges* option

is set to *true* by default. If *preventchanges=false* Acrobat enables all editing tools. However, modifications which are not allowed still invalidate the certification signature.

Since a certification signature must always be the first signature in a document it shouldn't be applied to a document which already contains a signature.

Table 7.6 Document modifications which are allowed without breaking a certification signature

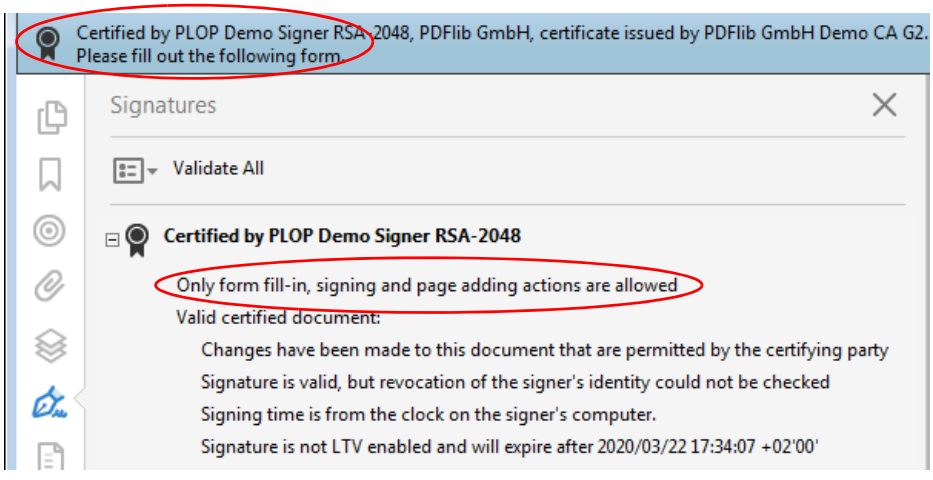
type of signature (option list)	***changes which are allowed without breaking the signature***				
	enter form field values	*digitally sign and add pages*[1]	*create, delete or modify annotations*	*add signature fields*[2]	*all other modifications*
`certification=nochanges`	–	–	–	–	–
`certification=formfilling`	*yes*	*yes*[3]	–	–	–
`certification=formsandannotations`	*yes*	*yes*[3]	*yes*	–	–
`certification=none` *(i.e. approval signature or document-level timestamp signature)*	*yes*	*yes*	*yes*	*yes*	–

1. Adding pages with a rarely used technique called spawning page templates is allowed, but not manually adding pages with `Tools, Pages, Insert Pages`.
2. Adding signature fields via `Fill&Sign, Place Signature` *is allowed, but not adding form fields with* `Tools, Forms, Edit`.
3. Only signing by clicking a signature field is allowed, but not via Acrobat's menu items.

Validity of certification signatures in Acrobat. Even if a certification signature is technically valid there are some additional requirements for fully leveraging the benefits of certified documents in Acrobat:

- Certification signatures are most easily created with certificates from an AATL CA (see »Trusted Root Certificates in Acrobat«, page 90). Since the Adobe Root CA automatically has the required trust setting no configuration steps are required.
- If end-user certificates under a root which is not known to Acrobat are intended to create certification signatures, it is recommended to assign the necessary trust level

Fig. 7.6 Certification signature with »form filling and signing allowed« in Acrobat



to the root certificate in Acrobat as follows:
Edit, Preferences..., Signatures, Identities & Trusted Certificates, More..., Trusted Certificates, select the root certificate, *Edit Trust*, and activate *Certified documents*.
As a result, all certification signatures created with certificates under the selected root are accepted as valid.

- For an individual certificate you can also set the required trust level. However, this is rather untypical and not recommended. Proceed as follows:
Open the *Signatures* pane, select the certification signature, *Certificate Details...*, select the signing certificate in the certificate chain (i.e. the one at the bottom of the list), open the *Trust* tab, click *Add to Trusted Certificates...*, click OK in the informative message dialog, and edit the Trust settings.

If you don't apply any of the methods described above, Acrobat flags the certification signature with a yellow triangle instead of the badge and adds the text »The signer's certificate has not been trusted for the purpose of creating Certified documents«.

pCOS. Certification signatures are reported in pCOS as *signaturefields[...]/sigtype=certification*. The kind of allowed changes can be queried with *signaturefields[...]/permissions* which returns one of the keywords *nochanges*, *formfilling*, or *formsandannotations*.

The pCOS pseudo object *signaturefields[...]/preventchanges* can be used to check whether Acrobat's user interface elements will be disabled to make sure that the certification signature cannot accidentally be invalidated by applying prohibited changes.

Signing certified documents where changes are prohibited. Since a certification signature must be the first signature in a document, additional signatures can usually be applied after the first one. However, sometimes the certification signature has been created with the »No changes are allowed« setting (in PLOP DS: *certification=nochanges)*. This kind of input document raises a conflict which must be resolved by the application developer: while the file prohibits any changes including signatures, the application wants to sign which would invalidate the certification signature (unlike approval signatures, which allow additional signatures in update mode without being invalidated). This is even true for signing in update mode since the certification doesn't allow any kind of changes. The conflict can be resolved in the following ways:

- Default behavior: the input is rejected since the existing certification signature has priority over the new signature.
- The certification signature is removed and the new signature applied. This can be achieved with the option *sacrifice=signatures* of *create_document( )*.

7.4 Certificate Revocation Information

A signature can optionally include information about the revocation status of the signing certificate. This information can be used by signature validation software to make sure that the certificate was still valid (has not been revoked) at the time of signature. Two different methods are available to achieve this.

In Acrobat DC you can check revocation information in the certificate viewer as follows: open the *Signatures* pane, right-click the signature and select *Show Signature Properties...*, *Show Signer's Certificate...*, and go to the *Revocation* tab (see Figure 7.7 and Figure 7.8).

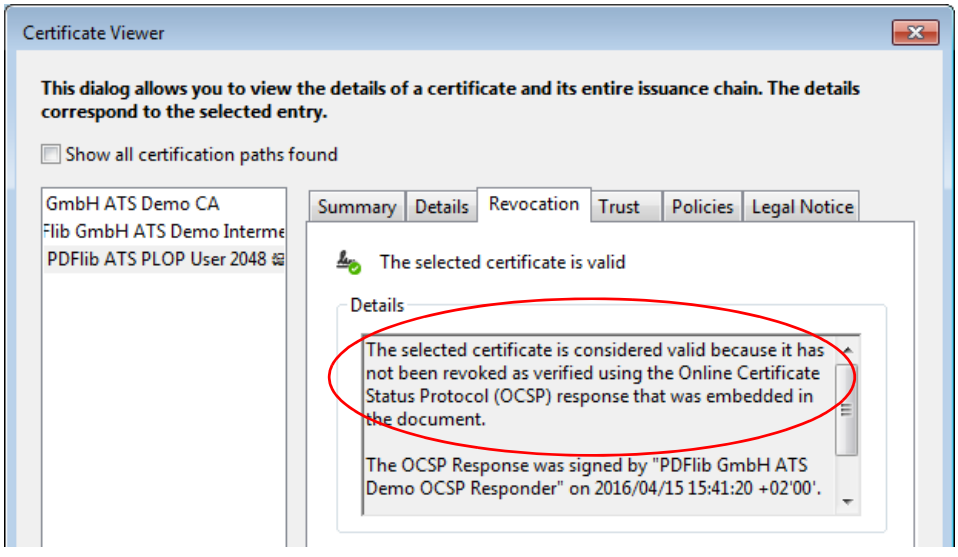
7.4.1 Online Certificate Status Protocol (OCSP)

Note *OCSP response embedding is not supported for* `engine=mscapi`.

OCSP overview. When OCSP according to RFC 2560 and RFC 6960 is used, the signing software sends a network request to an OCSP server (also called OCSP responder) to query the certificate's status in real-time. An OCSP responder is a server with real-time access to the CA's database of issued and revoked certificates. The OCSP responder checks whether the certificate is valid at the time of the query and returns a signed response with the result. This OCSP response is embedded in the signature.

A certificate may contain an extension called *Authority Info Access* (AIA) with the *ocsp* access method according to RFC 3280. This is usually the case for AATL certificates (see »Adobe Approved Trust List (AATL)«, page 90). This extension contains a URL for an OCSP responder associated with the CA which issued the certificate. The URL can alternatively be supplied via the *ocsp* signature option. When PLOP DS sends an OCSP request for a particular certificate, the OCSP server returns a signed response with the status *good*, *revoked,* or *unknown* for the certificate. In order to create a *good* OCSP response all of the following conditions must be met:

Fig. 7.7
OCSP information displayed in Acrobat



- An AIA extension with the *ocsp* access method must be present in the digital ID or the *source* suboption of the *ocsp* signature option must be supplied.
- The OCSP responder can be reached over the network at the specified URL and sends a response within the period specified in the *timeout* suboption of the *source* suboption of the *ocsp* signature option.
- The OCSP response contains the status *good* which requires that the certificate has been issued by the CA which is served by the OCSP responder, is valid (i.e. has not reached the end of its validity period) and has not been revoked.
- The signing time must lie in the interval defined by the entries *thisUpdate* and *nextUpdate* in the OCSP response, or (if *nextUpdate* is not present) the signing time must not be after *thisUpdate* plus the value specified in the *freshness* suboption. Both checks allow a tolerance up to the value of the *maxclockskew* suboption to compensate network delays or inaccurate system time.

If the OCSP response is *good*, PLOP DS embeds the response in the generated signature. Otherwise the unusable response is either ignored or no signature is created, depending on the *critical* suboption. By default PLOP DS uses the OCSP responder's URL in the AIA extension if present in the signer's digital ID and silently ignores situations without any good OCSP response. However, if OCSP response embedding is explicitly requested with the *ocsp* option a *good* response is required for generating a signature unless the *critical* option has been set to *false*.

OCSP configuration. Depending on the PKI in use you must consider the following configuration issues for OCSP responses:

- If no AIA extension is present in the certificate, you must supply an OCSP responder with the *source* suboption of the *ocsp* option.
- A valid certificate for the issuer of the signer's certificate is required for creating the OCSP request. It is often included in the signer's digital ID; otherwise it must be supplied separately with one the *rootcertdir/rootcertfile/certfile* signature options.
- Since the OCSP responder may require authentication for successful network communication several authentication options are supported for OCSP requests.
- The *nonce* feature of OCSP prevents replay attacks, but at the same time thwarts caching and therefore reduces performance. Depending on the configuration of the OCSP responder you may have to use the *nonce* option. If you get a message similar to the following the OCSP responder doesn't support the nonce feature. In this situation you can supply the signature option *nonce=false* to disable the nonce feature:

  ```
  OCSP response from URL 'http://ocsp.acme.com' for certificate 'CN = PDFlib GmbH...'
  does not contain nonce although it was requested
  ```

  The Microsoft OCSP responder will reject the request with an *unauthorized* error if it is not configured for nonce processing, but a nonce is requested. In this case you must also supply the signature option *nonce=false* to disable the nonce feature.
- The *hash* suboption of the *ocsp* option can be used to select a hash function which is used in the OCSP request and response to identify the certificate. However, Acrobat DC can only deal with OCSP responses which use the SHA-1 hash function, and cannot use OCSP responses with other hash functions for signature validation. For this reason other values than *sha1* require the signature option *conformance=extended*.

Revocation checking for the OCSP responder. The OCSP responder's signing certificate must be valid at the time of creating the OCSP response. In order to avoid a recursive problem (the OCSP responder's certificate would require another OCSP response) it is recommended to include the *id-pkix-ocsp-nocheck* extension according to RFC 2560 in the OCSP responder's certificate. This is true for almost all commercial OCSP responders. Alternatively, this certificate may contain the CRL distribution points *(CRLdp)* extension.

OCSP option list examples. In the examples below only the part of the option list which is relevant for OCSP response embedding is shown. Other signature options must be added as appropriate.

Try OCSP response embedding with the URL present in the signer's digital ID and fail with an error if no AIA extension with the *ocsp* access method is available in the digital ID:

```
ocsp={source={}}
```

or equivalently

```
ocsp={}
```

Request OCSP response embedding using the AIA extension if possible, but silently ignore any error:

```
ocsp={source={} critical=false}
```

or equivalently

```
ocsp={critical=false}
```

Don't embed an OCSP response even if the AIA extension is present in the digital ID:

```
ocsp=none
```

Explicitly provide the URL and a timeout of one second for the OCSP responder, overriding an entry in the AIA extension which may be present in the digital ID:

```
ocsp={source={url={http://ocsp.acme.com/} timeout=1000} }
```

Ensure that unsuccessful OCSP attempts prevent the signing process and disable the nonce feature for OCSP responders which don't support it:

```
ocsp={critical nonce=false}
```

7.4.2 Certificate Revocation Lists (CRLs)

Note CRL embedding is not supported for `engine=mscapi`.

CRL overview. When CRLs according to RFC 3280 and its successor RFC 5280 are used, the CA periodically (e.g. once per day) creates a signed list of certificates which haven't yet expired but have been revoked. This list is made available to the signature software and embedded in the signature. The list can be retrieved via the network or can be stored locally. CRLs have a specific lifetime (e.g. one day) and must be refreshed before the end of their lifetime. Since CRLs may cover any number of revoked certificates they

are typically much larger than OCSP responses (up to several megabytes), and their size is not known in advance. Since the full CRL is embedded in the PDF output this kind of revocation information bloats the signed PDF documents. PLOP DS can obtain CRLs from several sources:

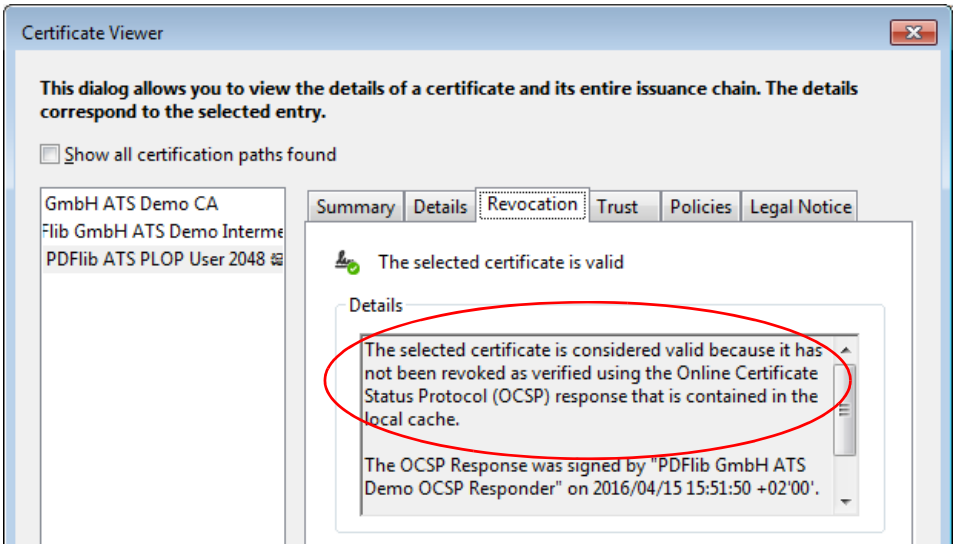
- A certificate may contain an extension called *CRL distribution points (CRLdp)*. This is always the case for AATL certificates (see »Adobe Approved Trust List (AATL)«, page 90). This extension contains one or more network URLs of CRL resources. PLOP DS tries all entries in the *CRLdp* extension until it can retrieve a CRL. If a usable CRL was found it is embedded in the signature or a Document Security Store (DSS) (see Section 7.3.3, »Document Security Store (DSS)«, page 106). The *CRLdp* extension is evaluated for each certificate for which is CRL is required, depending on the availability of an OCSP response and the respective *critical* options.
- As an alternative to the *CRLdp* extension, retrieval of a CRL for the signing certificate can be configured with the *crl* option. The suboption *source* points to a network address where a CRL is retrieved dynamically; the suboption *filename* points to a static local CRL file in DER encoding.
- One or more local CRL files for the signing certificate and all other involved certificates can be supplied in PEM encoding with the *crldir/crlfile* signature options.

If the signer's certificate is included in the CRL it has been revoked by the issuing CA, i.e. it can no longer be used to create a valid signature. In this case *prepare_signature( )* fails with an error message similar to the following:

```
Certificate verification failure for certificate with subject 'C = DE, L = Munich, O =
PDFlib GmbH, CN = PLOP Demo Signer RSA-2048': certificate revoked
```

PLOP DS uses CRLs until they expire. Only when a particular CRL can no longer be used because its lifetime has ended PLOP DS downloads a new CRL from the server.

Fig. 7.8
CRL information displayed in Acrobat



CRL option list examples. In the examples below only the part of the option list which is relevant for CRL embedding is shown. Other signature options must be added as appropriate.

Try CRL embedding with the URL present in the signer's digital ID and fail with an error if no *CRLdp* extension is available in the digital ID:

```
crl={source={}}
```

or equivalently

```
crl={}
```

Request CRL embedding using the *CRLdp* extension if possible, but silently ignore errors:

```
crl={source={} critical=false}
```

or equivalently

```
crl={critical=false}
```

Don't try to retrieve a CRL for the signing certificate or any other certificate even if the *CRLdp* extension is present in the digital ID. This makes sense if online retrieval is doomed to fail anyway, e.g. because the signing computer is offline:

```
crl=none
```

Explicitly provide the URL and a timeout of one second for the CRL server, overriding an entry in the *CRLdp* extension which may be present in the digital ID:

```
crl={source={url={http://crl.acme.com/} timeout=1000} }
```

Provide a CRL contained in a local disk file:

```
crlfile={certs.pem}
```

7.4.3 OCSP or CRL?

The following factors are relevant for selecting the most suitable method of including revocation information:

- OCSP provides real-time certificate status information. Since an OCSP response covers only a single certificate it has a predictable size of only a few kilobytes. On the other hand, OCSP always requires a network connection to the OCSP responder.
- The advantage of CRLs over OCSP is that they can be stored locally and therefore network overhead can be avoided. The disadvantage is that a locally stored CRL may become stale unless it is renewed frequently (i.e. published and downloaded).
- Since CA certificates rarely need to be revoked, CRLs for CAs are typically much smaller than CRLs for end-user certificates.
- Similarly, since HSMs are rarely broken or stolen, CRLs for an HSM-based certificate are often usually very small (provided the CRL covers only HSM-based certificates and no file-based certificates).
- Some legislations or private signature policies may require or prohibit one of both methods.

By default, PLOP DS only embeds a CRL in the signature if no valid good OCSP response is available, but this behavior can be modified with the *ocsp* and *crl* options and the *critical* suboption.

Use the following option list to ensure that revocation information is always embedded, where a CRL will only be retrieved if OCSP doesn't provide a good response:

```
ocsp={critical=false source={url={http://ocsp.acme.com/}}} ↵
 crl={critical=true  source={url={ http://crl.acme.com/}}}
```

Keep in mind that the *ocsp* and *crl* options control only revocation information embedding for the signing certificate, but not for any CA or TSA certificates which may be involved.

7.5 Timestamps

7.5.1 Timestamp Configuration

A digital signature may include date and time information obtained from a trusted time server, also called Timestamp Authority (TSA). Unlike the time taken from the signing computer (which can easily be manipulated), a timestamp obtained from a trusted server provides a signed and reliable source for the time of signature. PLOP DS supports timestamping according to RFC 3161, RFC 5816, and ETSI EN 319 422. Since the timestamping request includes a hash of the generated signature, the timestamp confirms that the signature has been created at a particular time. The timestamp is embedded in the generated PDF signature.

PLOP DS validates each timestamp before embedding it in the generated document. An error is raised if a requested timestamp cannot be validated e.g. the required credentials have not been configured. The user is responsible for selecting and configuring a suitable TSA, e.g one which creates qualified timestamps.

Depending on the selected TSA you must consider the following configuration issues for creating timestamps:

- The most important information is the network address where the TSA can be reached. It can be supplied with the *url* suboption of the *source* suboption. Alternatively it can be taken from the signer's digital ID (see »Timestamp extension in digital ID«, page 119).
- In order to trust the TSA the CA which issued the TSA certificate must be trusted. The TSA's CA certificate must be handled the same way as other CA certificates when validating a signature; see »Configuring trust root certificate(s) for all chains«, page 125, for details. This is especially important for creating LTV-enabled signatures. If you work with a TSA under one of the AATL hierarchies (see »Adobe Approved Trust List (AATL)«, page 90) the issuer or chain of issuers of the TSA certificate is known to Acrobat as a trusted root. However, it may be necessary to supply the TSA CA certificate to PLOP DS in the *certfile* option.
- The TSA may require the client to use a particular hash algorithm for creating the timestamp request. By default, PLOP DS uses the SHA-256 algorithm which works with all modern TSAs. Another hash function can be supplied with the *hash* suboption. Note that the hash algorithm used in the timestamp signature cannot be specified since it is completely under control of the TSA.
- While some TSAs are freely accessible, commercial TSAs may require user name and password to restrict access. Unauthorized access results in a message similar to the following:

  ```
  Network response from URL 'https://timestamp.acme.com/tsa' has bad status code 401
  ('Unauthorized')
  ```

  Authentication parameters can be supplied as part of the URL or with the suboptions *username/password* of the *source* network suboption.
- If the TSA requires SSL access (i.e. *https)* the server's SSL root certificate must be supplied with the *sslcertdir/sslcertfile* options. Otherwise you will run into a message similar to the following:

  ```
  Document timestamp request to 'https://timestamp.acme.com/tsa' failed
  ('Peer certificate cannot be authenticated with given CA certificates')
  ```

As an alternative to providing the required server certificate you can skip the server certificate check with the option *sslverifypeer=false*, provided you are aware of the security implications.

- Some TSAs require an explicit policy OID (object identifier) which can be supplied with the *policy* suboption. The applicable value of the OID must be arranged with the TSA. The policy OID is displayed in Acrobat's *Signature Properties* dialog, *Advanced Properties...* .

7.5.2 Timestamped Signatures

Note *Timestamped signatures are not supported for* `engine=mscapi`.

Approval and certification signatures may optionally contain an embedded timestamp. Timestamped signatures are supported in Acrobat 7 and above.

Timestamp extension in digital ID. A digital ID may contain the *TimeStamp* extension which contains the URL of a timestamp authority to facilitate signing with embedded timestamps without the need for supplying TSA details. The *TimeStamp* extension is usually included in certificates issued by AATL *(Adobe Approved Trust List)* providers (see »Adobe Approved Trust List (AATL)«, page 90). Some AATL providers offer a limited number of timestamps for free.

If the *TimeStamp* extension is present and contains a URL which does not require authentication, PLOP DS attempts to access the specified TSA for creating a timestamp. In this situation it is not necessary to supply the *url* suboption for creating a timestamp. In order to use a TSA which requires authentication, however, you must specify the full TSA details explicitly in an option list (see examples below), even if the TSA is specified in the *TimeStamp* extension.

Timestamping option list examples. In the examples below only the part of the option list which is relevant for including a timestamp is shown. Other signature options must be added as appropriate.

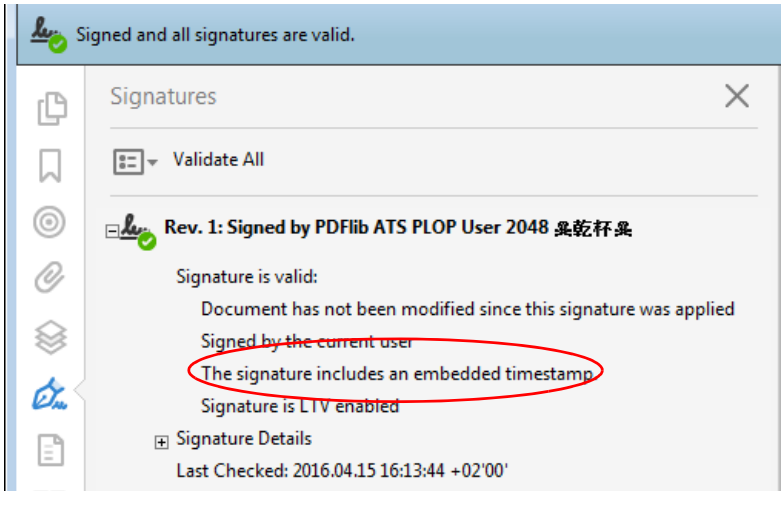


Fig. 7.9 Timestamped signature in Acrobat

Stamp the signature with a timestamp obtained from the TSA at the specified URL using the default hash algorithm SHA-256:

```
timestamp={source={url={http://timestamp.acme.com/tsa}}}
```

Stamp the signature with a timestamp where the TSA requires user name and password to obtain a timestamp:

```
timestamp={source={url={http://timestamp.acme.com/tsa} username=demo password=demo}}
```

Stamp the signature with a timestamp where the TSA requires digest authentication:

```
timestamp={source={url={http://timestamp.acme.com/tsa} httpauthentication=digest ↵
        username=demo password=demo }}
```

If the TSA must be accessed via SSL you must supply the server's SSL certificate with the options *sslcertdir/sslcertfile*. If the server's SSL certificate is not available you can skip server authentication with the *sslverifypeer* option, provided you are aware of the security implications of doing so:

```
timestamp={source={url={https://timestamp.acme.com/tsa}} sslverifypeer=false}
```

Attempt to embed a timestamp in the signature with the URL present in the signer's digital ID and fail with an error if no *TimeStamp* extension is available in the digital ID:

```
timestamp={source={}}
```

or equivalently

```
timestamp={}
```

Don't embed a timestamp even if the *TimeStamp* extension is present in the digital ID:

```
timestamp=none
```

7.5.3 Document-Level Timestamp Signatures

Document-level timestamps have been introduced with PAdES part 4 and are included in ISO 32000-2.

Timestamped signature vs. document-level timestamp. Similar to a timestamped signature a document-level timestamp provides status information related to a particular point in time. However, in the first case the timestamp is an attribute of the main signature, while a document-level timestamp is a valid signature of its own. It does not require any digital ID since no signing person or entity is involved. Instead, document-level timestamps are created via a network request to a Timestamp Authority (TSA). Document-level timestamps ensures that a particular document has been in existence at the time designated in the timestamp.

Note Document-level timestamp signatures are not supported for `engine=mscapi`.

Document-level timestampings option list examples. In the examples below the full signature option list for creating a document timestamp is shown. Since no signing certificate is required no other signature options are required.

Add a document-level timestamp obtained from the TSA at the specified URL using the default hash algorithm SHA-256:

```
doctimestamp={source={url={http://timestamp.acme.com/tsa}}}
```

Add a document-level timestamp from a TSA which requires user name and password:

```
doctimestamp={source={url={http://timestamp.acme.com/tsa}} username=demo password=demo}
```

Add a document-level timestamp from a TSA which requires digest authentication:

```
doctimestamp={source={url={http://timestamp.acme.com/tsa} httpauthentication=digest ↵
        username=demo password=demo}}
```

pCOS. Document-level timestamp signatures are reported in pCOS as *signaturefields[...]/sigtype=doctimestamp*.

7.5.4 Troubleshooting and Unsupported TSA Types

Oversized timestamp responses. PLOP DS must know the size of a timestamp in advance. It uses a built-in value for the maximum size of the timestamp response received from the TSA. If the timestamp response exceeds this maximum the following error occurs:

```
Not enough space reserved for signature contents (reserved XXX bytes, need YYY bytes)
```

In this case you can increase the maximum value with the signature option *timestampsize*. The internal default value of *timestampsize* is documented in Table 8.7, page 153.

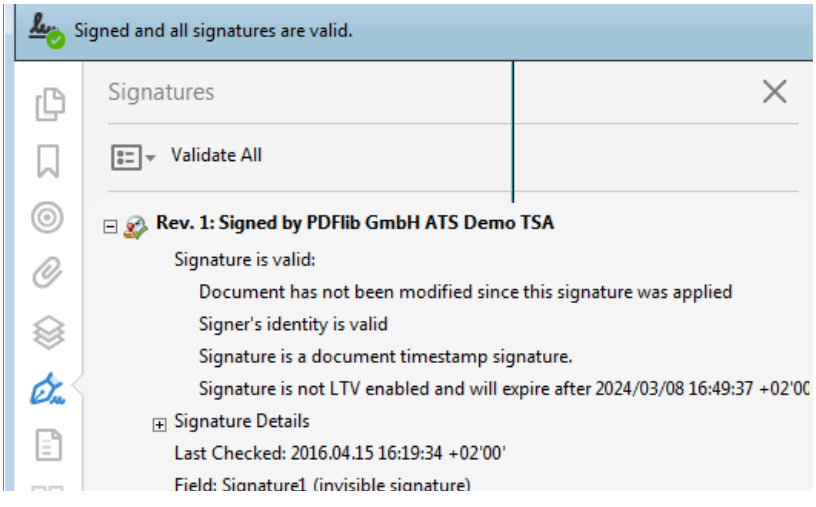


Fig. 7.10 Document-level timestamp in Acrobat

In the situations described below a TSA cannot be used for signing PDF documents with PLOP DS.

Attribute certificates. Attribute certificates are not supported in PLOP DS. If a TSA uses them PLOP DS issues the following error message:

```
Timestamp authority 'http://adobe-tsa.entrust.net/TSS/HttpTspServer'
uses unsupported protocol ('wrong tag')
```

A particular application of attribute certificates is for a TSA's Time Auditing Certificate (TAC). Some TSA products use the new CMS syntax according to RFC 2630 for encoding the TAC which is not supported in PLOP DS. However, they can be configured to encode the TAC with alternative methods such as putting the TAC in a signed attribute according to RFC 3126.

Missing »critical« flag in key usage extension. The timestamping protocol RFC 3161 requires that the TSA certificate includes the *Extended Key Usage* extension with the value *timestamping*, where this extension must be marked as critical. If this extension is present in the TSA certificate, but not marked as critical, Acrobat rejects the signature as invalid.

PLOP DS rejects timestamps produced with such a TSA certificate with the following error message:

```
Signature verification of timestamp failed: certificate verify error:
Verify error:unsupported certificate purpose
```

Trying to use a TSA certificate which doesn't have the »critical« flag set for the *Extended Key Usage* field to create a document timestamp with Acrobat results in the following error message:

```
Error encountered while signing:
Certificate is not valid for the usage
```

Using such a TSA to create a certification or approval signature with an embedded timestamp with Acrobat succeeds, but upon validation the timestamp in the resulting signature is rejected with the following message:

```
The signature includes an embedded timestamp but it is invalid
```

Authenticode Timestamp servers. Authenticode is a Microsoft timestamping protocol which had its main use for code-signing. Since Authenticode is based on the older RFC 2985/PKCS#9 instead of RFC 3161 it is not supported in PDF and PLOP DS.

PLOP DS rejects timestamps produced with an Authenticode TSA with an error message similar to the following:

```
Unexpected content type 'text/html;charset=ISO-8859-1' in reply to timestamp request to
URL 'http://timestamp.entrust.net/TSS/AuthenticodeTS'
(expected content type 'application/timestamp-reply')
```

or

```
Unexpected content type 'application/timestamp-query' in reply to timestamp request to
URL 'http://timestamp.verisign.com/scripts/timstamp.dll'
(expected content type 'application/timestamp-reply')
```

Trying to use an Authenticode TSA with Acrobat results in the following error message:

```
Error encountered while signing:
Error encountered while BER decoding
```

7.6 Long-Term Validation (LTV)

7.6.1 LTV Concept and Acrobat Support

Long-Term Validation (LTV) means that a signature can still be validated once the signing certificate has expired or has been revoked, which is an important aspect for archiving signed documents over long periods of time. The LTV concept is discussed in PAdES Part 4 (ETSI TS 102 778-4) and supported in Acrobat DC. LTV signatures conform to the requirements of the eIDAS regulation.

In order to LTV-enable a signature the full certificate chain and revocation information for all involved certificates, collectively called validation information, must be embedded in the signature or in a DSS (see Section 7.3.3, »Document Security Store (DSS)«, page 106). Since more signature-related data must be embedded for LTV, the signed documents are typically larger than non-LTV-enabled signatures.

Note *LTV-enabled signatures are not supported for* `engine=mscapi`.

LTV-enabled signatures should include an embedded timestamp, but this is not a strict requirement. You can extend the lifetime of an LTV-enabled signature by adding a document-level timestamp signature before any of the involved certificates expires or is revoked.

LTV status is not defined in absolute terms, but relative to a set of trusted root certificates. Depending on configuration, a particular signature may be regarded as LTV-enabled in one configuration, but not LTV-enabled in another one. For example, if you configure different trusted roots in PLOP DS and in Acrobat the LTV status may be different.

LTV status in Acrobat. Acrobat DC displays the status line »Signature is LTV enabled« or »Signature is not LTV enabled and will expire after...« in the Signatures pane (see Figure 7.11). Keep the following in mind related to the LTV status line:

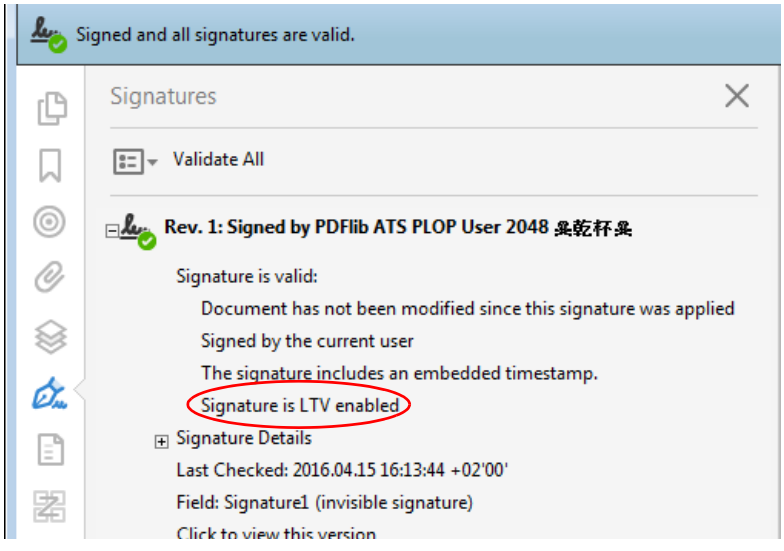


Fig. 7.11
LTV-enabled signature in Acrobat

- The root certificate(s) for all involved certificates must be configured as trusted in Acrobat (see »Trusted Root Certificates in Acrobat«, page 90).
- Any valid signature can be forced to display as LTV-enabled in Acrobat by adding the immediate signing CA certificates to the trusted root store. This may cause confusion regarding the LTV status if the configuration is not taken into account. It also implies that signatures created with a self-signed certificate are treated as LTV-enabled if the certificate is added to the trusted root certificates.
- Acrobat DC does not insist on an embedded timestamp to achieve LTV status. If a timestamp is embedded, Acrobat does not require validation information for the TSA certificate. PLOP DS is stricter and also requires full validation information for the TSA certificate. Acrobat doesn't use an embedded OCSP response for the TSA certificate unless it has been created only a few minutes before validation time.
- Acrobat DC supports only the SHA-1 hash function for OCSP responses (see »OCSP configuration«, page 113). As a result, Acrobat may not display the LTV status correctly if another hash function is used although full validation information is actually available.
- The following setting must not be activated in Acrobat since otherwise the LTV status is not shown: *Preferences, Signatures, Verification, More..., Verification Time, Verify Signatures Using: Current time.*
- The LTV status may get lost by reverting to an earlier revision; see »Reverting to earlier revisions of a signed document«, page 108, for details.

7.6.2 LTV-enabled Signatures with PLOP DS

If the following option is supplied, PLOP DS creates an LTV-enabled signature provided all validation information can be obtained. Otherwise an error is issued and no signature is created:

```
ltv=full
```

This option alone does not ensure LTV-enabled signatures, but only checks that all requirements are met. If a certificate is missing or validation information could not be obtained, PLOP DS issues an error message. It it therefore important to thoroughly analyze all error messages.

The default setting *ltv=try* means that all available revocation information is embedded in the signed document, but the signature call does not fail if the validation information is not sufficient for achieving LTV status.

Configuring trust root certificate(s) for all chains. In order to fully validate all involved certificates PLOP DS needs trust anchors for all certificates. The exact number depends on the PKI configuration. Trusted root certificates must be supplied with the *rootcertdir* or *rootcertfile* option. This involves at least the certificate of the root CA at the top of the chain for the signing certificate. Other root certificates, e.g. for the TSA, may be required unless a single root CA is at the top of all involved certificate chains.

Configuring intermediate CA certificate(s). The remaining certificate chain (i.e. all intermediate CAs between the root and the signing certificate or other involved certificates) must be available so that PLOP DS can embed it in the signature. CA certificates for the signing certificate as well as other involved certificates such as an OCSP responder certificate or TSA certificate are searched in the following locations:

- CA certificates can be supplied with the *certfile* option.
- (Not for *engine=mscapi)* CA certificates for the signing certificate can be included in the PKCS#12 file which contains the signer's digital ID.
- (Only for *engine=mscapi)* CA certificates are searched in the Windows certificate store.
- A certificate may contain the *Authority Info Access* (AIA) extension with the *caIssuers* (Certification Authority Issuer) access method according to RFC 3280. This extension contains one or more URLs where the certificate of the CA which issued the signing certificate and possibly intermediate CA certificates can be downloaded. The protocols *http*, *https*, and *ftp* are supported.

A certificate may specify the LDAP protocol in the AIA extension which is not currently supported in PLOP DS. In this situation you can use an LDAP browser[1] to retrieve the CA certificate manually via LDAP and supply it to the options mentioned above. This must be done only once for a signing certificate.

Which CA certificates do I have to configure? The detailed requirements for achieving LTV status depend on the PKI configuration. In many cases the following steps are sufficient:

- Certificates issued by many commercial CAs include the AIA extension with the *caIssuers* access method. This means that PLOP DS can automatically download the chain of CA certificates for the signing certificate. Only the root CA certificate must be supplied with the *rootcertdir* or *rootcertfile* option.
  If the AIA extension with the *caIssuers* access method is not present in the signing certificate you can usually download the required root certificate(s) from the CA's Web site.
- Certificates of TSAs and OCSP responders are retrieved automatically. If these certificates or any intermediate certificates have been issued by another root CA than the signing certificate, the root certificate must be supplied with the *rootcertdir* or *rootcertfile* option.
- CRLs are often signed by the same CA which issued the certificate that is being queried. However, if the CRL has been signed by another CA the corresponding CA certificate must be supplied with the *certfile* option since it cannot be downloaded automatically. If the certificate used for signing a CRL has been issued by another root CA than the signing certificate (e.g. the CRL for a TSA which is based in another PKI), the root certificate must be supplied with the *rootcertdir* or *rootcertfile* options.

With *validate=full* or *ltv=full* PLOP DS emits an error »*unable to get local issuer certificate*« if a required CA certificate is missing. In this case you must supply the missing certificate in one of the options *rootcertdir*, *rootcertfile, or certfile*. The following message:

```
Certificate verification failure for certificate with subject '...':
self signed certificate in certificate chain
```

typically occurs if you supplied a trusted self-signed certificate in the *certfile* option instead of the *rootcertfile* or *rootcertdir* option.

1. For example the free Softerra LDAP Browser which is available from www.ldapbrowser.com/

Revocation information for the signing certificate. Revocation information for the signing certificate must be supplied by one of the following means:

- OCSP via the *AIA* extension in the signer's certificate or the *ocsp* option.
- CRL via the *CRLdp* extension in the signer's certificate or the *crl* option. Existing CRLs can be supplied with the *crldir* and *crlfile* options.

The *critical* suboption of the *ocsp* and *crl* options can be used to make sure that a signature is only created if OCSP or CRL information for the signing certificate could be obtained successfully. See Section 7.4, »Certificate Revocation Information«, page 112, for details.

Revocation information for other involved certificates. Revocation information for the certificates of all CAs in the certificate chain as well as for the certificates of all CAs used for signing CRLs and OCSP responses must also be available, with the following exceptions which don't require revocation information:

- root CA certificates supplied to the *rootcertdir* or *rootcertfile* options;
- an OCSP responder's certificate if it includes the *id-pkix-ocsp-nocheck* extension (which is commonly the case).

Revocation information for certificates other than the signing certificate can be provided by one of the following means:

- OCSP via the *AIA* extension in the certificate.
- CRL via the *CRLdp* extension in the certificate. Existing CRLs can be supplied with the *crldir* and *crlfile* options.

LTV option list examples. For the first example let's assume that the PKI is set up as follows:

- the signer's digital ID contains the chain of CA certificates in the PKCS#12 file, or each certificate except the root certificate contains an AIA extension with *caIssuers* access method;
- the signer's digital ID and all CA certificates in the chain except the root certificate contain an AIA extension with the *ocsp* access method or *CRLdp* extension;
- the OCSP responder's certificate contains the *id-pkix-ocsp-nocheck* extension.

In this situation only the *rootcertfile* option is required (in addition to options for the digital ID) to achieve LTV status. The option *ltv=full* can be used to ensure that violations of LTV requirements are detected and no signature is created if LTV status cannot be achieved:

```
digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ltv=full ↵
        rootcertfile=root1.pem
```

In order to embed a timestamp, revocation information for the TSA certificate must also be available to achieve LTV status. Ideally, the TSA certificate also contains the AIA extension with *ocsp* access method or *CRLdp* extension and is rooted in the same CA as the signing certificate. In this case no more specific options are required to achieve LTV status:

```
digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ltv=full ↵
        rootcertfile=root1.pem timestamp={source={url={http://timestamp.acme.com/tsa}}}
```

However, if the TSA is based on a different root CA you must also supply the TSA root in the *rootcertfile* option (the file *root1+2.pem* is assumed to contain both required root certificates in PEM encoding):

```
digitalid={filename=demo_signer_rsa_2048.p12} passwordfile=pw.txt ltv=full ↵
        rootcertfile=root1+2.pem timestamp={source={url={http://timestamp.acme.com/tsa}}}
```

7.7 The CAdES and PAdES Signature Standards

7.7.1 CMS and CAdES Signatures

The European Telecommunications Standards Institute (ETSI)[1] issued a number of digital signature standards to harmonize digital signatures among EU member countries. ETSI standards are quite influential also in other parts of the world. They are referenced in the PDF 2.0 standard ISO 32000-2 and have been incorporated in various RFCs.

CMS and CAdES signatures. For a long time PDF signatures were based on CMS (Cryptographic Message Syntax). It is specified in RFC 5652 and widely used in Internet protocols. CMS signatures in PDF use the subfilter *adbe.pkcs7.detached* or some older deprecated entry in the signature dictionary. CMS signatures can be created and validated with all Acrobat versions.

CAdES *(CMS Advanced Electronic Signatures)* is specified in ETSI TS 101 733 (technically equivalent to RFC 5126) and adds some features to CMS. Most importantly, it protects against a threat scenario called certificate substitution by including a reference to the signing certificate in the signature (using the *signing-certificate-v2* attribute). CAdES signatures in PDF require the *ETSI.CAdES.detached* subfilter in the signature dictionary.

pCOS. CAdES signatures are reported in pCOS as *signaturefields[...]/cades=true*.

PAdES signatures. PAdES *(PDF Advanced Electronic Signatures)* is specified in ETSI TS 102 778. It applies CAdES to PDF by adding options and constraints to PDF signatures as defined in PDF 1.7 (ISO 32000-1). PAdES also specifies additional PDF data structures which are included in PDF 2.0 (ISO 32000-2). PAdES consists of several parts (those parts which are not relevant here are omitted).

- PAdES Part 2 is specified in ETSI TS 102 778-2 PAdES Basic. It is based on CMS and conforms to ISO 32000-1 while forbidding some of its optional features to strengthen the signatures. For example, PAdES-Basic requires the *ByteRange* to cover the whole document in order to close a security gap in the original ISO 32000-1 definition.
- PAdES Part 3 is specified in ETSI TS 102 778-3 PAdES Enhanced. It is based on CAdES and defines the two profiles BES *(Basic Electronic Signature)* and EPES *(Explicit Policy-based Electronic Signature)*. EPES extends BES by adding a policy identifier and an optional commitment type indication to the signature. The *policy* attribute specifies the signature policy under which the signature is created. The *commitment-type* attribute can be used as an alternative to the *Reason* entry in the signature dictionary. CAdES defines a series of generic commitment types such as *proof of origin*, *proof of receipt*, or *proof of approval*.
- PAdES Part 4 is specified in ETSI TS 102 778-4 PAdES Long-Term and provides the means for long-term validation, discussed in more detail in Section 7.6, »Long-Term Validation (LTV)«, page 124. Part 4 introduces document-level timestamps and the DSS (see Section 7.3.3, »Document Security Store (DSS)«, page 106). The concepts defined in PAdES part 4 can be applied to PAdES part 2 and part 3 signatures, i.e. PAdES-LTV can be based on CMS or CAdES.

1. ETSI standards are freely available from www.etsi.org/standards

PAdES signature levels. Several levels of PAdES signatures have been defined which are intended to cover the life cycle of a signature. The following PAdES basic signature levels are defined in ETSI EN 319 142-1:

- Basic Signature: PAdES Level B-B is the building block of PAdES signatures. It supports signatures with or without signature policy identifier, i.e. EPES and BES. This level can be considered a profile for short-term signatures.
- Signature with Time: PAdES Level B-T adds a signature timestamp to PAdES Level B-B to prove that the signature existed at a certain date and time.
- Signature with Long-Term Validation Material: PAdES Level B-LT adds validation data to PAdES Level B-T to ensure long-term availability of the validation material. Since all certificates of the certificate chain as well as OCSP responses or CRLs are available the signature can be validated even after a long time, e.g. when the CA is no longer available.
- Signature providing Long Term Availability and Integrity of Validation Material: PAdES Level B-LTA adds a document-level (archive) timestamp and associated validation data to PAdES Level B-LT to ensure long-term availability and integrity of the validation material. Timestamps can be added repeatedly as required, e.g. when cryptographic algorithms or key lengths are no longer considered strong enough. LTA signatures conform to the requirements of the eIDAS regulation.

ETSI EN 319 142-2 »Part 2: Extended PAdES signatures« defines extended signature profiles:

- PAdES Level E-BES specifies the basic requirements for digital signatures in PDF.
- PAdES Level E-EPES is built on PAdES E-BES. It adds a signature policy identifier and an optional commitment type indication.
- PAdES Level E-LTV builds on either E-BES or E-EPES. It adds a Document Security Store (DSS) and document timestamps. This can be used to augment an existing signature to maintain the long-term validity of the signature.

CAdES and PAdES support in Acrobat. By default Acrobat signatures conform to PAdES part 2 (PAdES-Basic). Acrobat DC can create PAdES part 3 BES signatures if configured for CAdES as follows: *Edit, Preferences, Signatures, Creation & Appearance, More..., Default Signing Format: CAdES-Equivalent*

Since there is no support for policy identifiers, PAdES E-EPES cannot be created with Acrobat. However, both E-BES and E-EPES can be validated with Acrobat.
PAdES part 4 is supported in Acrobat DC with the following features:

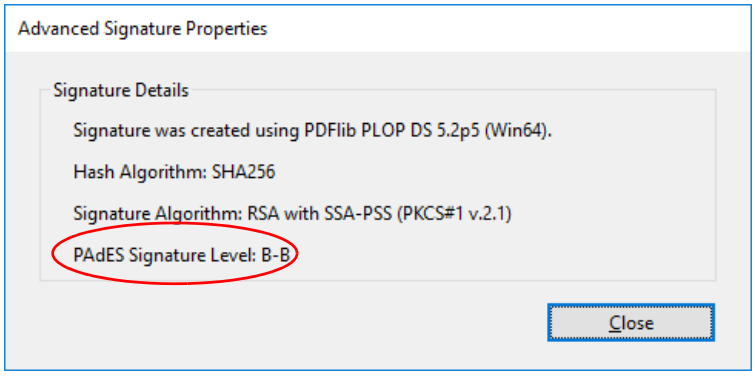


Fig. 7.12
PAdES status in Acrobat DC

- Long-Term Validation status information, see »LTV status in Acrobat«, page 124, for details;
- LTV-enabling a signature by opening the *Signatures* pane and clicking *Add Verification Information* in the *Options* menu.
- document-level timestamps.

Acrobat DC displays the PAdES signature level by right-clicking on a signature and selecting *Show Signature Properties..., Advanced Properties...*(see Figure 7.12).

7.7.2 PAdES Signatures with PLOP DS

PLOP DS supports all PAdES signature levels mentioned above for author and approval signatures. The signature type CMS or CAdES can be selected with the *sigtype* option; features for PAdES signature levels are activated by additional options. By default, PLOP DS creates CAdES signatures which conform to PAdES Level B-B. Table 7.7 lists the options required to achieve the PAdES signature levels with PLOP DS.

Note PAdES part 3 and part 4 are not supported for `engine=mscapi`.

Table 7.7 PAdES signature levels according to ETSI EN 319 142-1

PAdES signature level	***PLOP DS options***
PAdES Level B-B, includes E-BES (Basic Electronic Signature) and E-EPES (Explicit Policy Electronic Signature)	*PAdES E-BES:* `sigtype=cades` *(default)* *PAdES E-EPES: like PAdES E-BES plus* `policy`
PAdES Level B-T (Trusted time for signature existence)	*like PAdES Level B-B plus* `timestamp` *with* `critical=true`
PAdES Level B-LT (Long Term)	*like PAdES Level B-T plus* `ltv=full rootcertfile/rootcertdir`[1]
PAdES Level B-LTA (Long Term with Archive timestamps) (Required for qualified eIDAS signatures)	*like PAdES Level B-LT plus* `doctimestamp`
PAdES Level E-LTV (Long Term Validation)	*like PAdES Level B-LTA plus* `dss=true`*, applied to a document with an existing signature*

1. Additional options may be required to achieve LTV status, such as `certfile`, `ocsp`, `crl`*; see Section 7.6.2, »LTV-enabled Signatures with PLOP DS«, page 125.*

PAdES option list examples. The following signature option (in addition to other relevant options such as *digitalid)* creates a signature according to PAdES E-BES (since this is the default setting it can also be omitted):

```
sigtype=cades
```

The following partial signature option list creates a signature according to PAdES E-EPES (using a fictitious signature policy identifier):

```
policy={oid=2.16.276.1.89.1.1.1.1.3 commitmenttype=origin}
```

The following partial signature option list creates a signature according to PAdES Level B-T:

```
timestamp={critical source={url={http://timestamp.acme.com/tsa}}}
```

The following partial signature option list creates a signature according to PAdES Level B-LT:

```
timestamp={critical source={url={http://timestamp.acme.com/tsa}}} ltv=full
```

The following partial signature option list creates a timestamped signature with an additional archival timestamp according to PAdES Level B-LTA:

```
ltv=full timestamp={critical source={url={http://timestamp.acme.com/tsa}}} ↵
doctimestamp={source={url={http://timestamp.acme.com/tsa}}}
```

The following partial signature option list can be used to augment an existing PAdES Level B-LT signature with an additional archival timestamp according to PAdES Level B-LTA:

```
ltv=full doctimestamp={source={url={http://timestamp.acme.com/tsa}}}
```

8 PLOP and PLOP DS Library API Reference

8.1 Option Lists and other General Topics

8.1.1 Option List Syntax

Option lists are a powerful yet easy method to control PLOP operations. Instead of requiring a multitude of function parameters, many API methods support option lists, or optlists for short. These are strings which may contain an arbitrary number of options. Optlists support various data types and composite data like arrays. In most languages optlists can easily be constructed by concatenating the required keywords and values. C programmers may want to use the *sprintf( )* function in order to construct optlists. An optlist is a string containing one or more pairs of the form

```
name value(s)
```

Names and values, as well as multiple name/value pairs can be separated by arbitrary whitespace characters (space, tab, carriage return, newline). The value may consist of a list of multiple values. You can also use an equal sign '=' between name and value:

```
name=value
```

Simple values. Simple values may use any of the following data types:

- Boolean: *true* or *false*; if the value of a boolean option is omitted, the value *true* is assumed. As a shorthand notation *noname* can be used instead of *name=false*.
- String: strings containing whitespace or '=' characters must be bracketed with *{* and *}*. An empty string can be constructed with *{ }*. The characters *{*, *}*, and \ must be preceded by an additional \ character if they are supposed to be part of the string.
- Text strings are a special kind of string for certain options. While most options of type string accept only ASCII values, text strings may also carry Unicode values beyond ASCII. In Unicode-capable language bindings (see »Unicode Support in Language Bindings«, page 137) you can simply supply arbitrary Unicode values for such options. In non-Unicode-capable language bindings the user must prepend a UTF-8 BOM to text strings if the string is to be interpreted as UTF-8. If no UTF-8 BOM is present, text strings will be interpreted in *auto* encoding, i.e. the current code page on Windows, *ebcdic* on IBM Z, and *iso8859*-1 on Unix and macOS.
- Keyword: one of a predefined list of fixed keywords
- Float and integer: decimal floating point or integer numbers; point and comma can be used as decimal separators.
- Handle: several internal object handles, e.g., document or page handles. Technically these are integer values.

Depending on the type and interpretation of an option additional restrictions may apply. For example, integer or float options may be restricted to a certain range of values; handles must be valid for the corresponding type of object, etc. Conditions for options

are documented in their respective function descriptions. Some examples for simple values (the first line shows a string containing a blank character):

```
password={secret string}
linearize=true
```

List values. List values consist of multiple values, which may be simple values or list values in turn. Lists are bracketed with *{* and *}*. Example for a list value:

```
permissions={ noprint nocopy }
```

Note The backslash \ character requires special handling in many programming languages

Unquoted string values. In the following situations the actual characters in an option value may conflict with optlist syntax characters:

- Passwords or file names may contain unbalanced braces, backslashes and other special characters
- Japanese SJIS filenames in option lists (reasonable only in non-Unicode-capable language bindings)

In order to provide a simple mechanism for supplying arbitrary text or binary data which does not interfere with option list syntax elements, unquoted option values can be supplied along with a length specifier in the following syntax variants:

```
key[n]=value
key[n]={value}
```

The decimal number *n* represents the following:

- in Unicode-capable language bindings: the number of UTF-16 code units
- in non-Unicode aware language bindings: the number of bytes comprising the string

The braces around the string value are optional, but strongly recommended. They are required for strings starting with a space or other separator character. Braces, separators and backslashes within the string value are taken literally without any special interpretation.

Example for specifying a 7-character password containing space and brace characters. The whole string is surrounded by braces which are not part of the option value:

```
password[7]={ ab}c d}
```

Rectangle. A rectangle is a list of four float values specifying the *x* and *y* coordinates of the lower left and upper right corners of a rectangle. The coordinates are interpreted in the default PDF coordinate system, i.e. origin in the lower left corner of the page and point as unit. Example:

```
rect={ 100 100 200 150}
```

The *adapt* keyword can be used for automatic size calculation without distortion, see Section 7.3.1, »Visualizing Signatures with a Graphic or Logo«, page 102.

8.1.2 Unicode Support in Language Bindings

If a programming language or environment supports Unicode strings natively we call the binding Unicode-capable. The following language bindings are Unicode-capable:

- C++
- .NET
- Java
- Objective-C
- Python
- RPG

String handling in these environments is straightforward: all strings are supplied as Unicode strings in native UTF-16 format. The language wrappers correctly deal with Unicode strings provided by the client and automatically set certain options.

The following language bindings are not Unicode-capable by default:

- C (no native string data type available)
- Perl
- PHP
- Ruby

The use of UTF-8 is recommended for non-Unicode-capable language bindings. Some aspects of the API differ between Unicode-capable and non-Unicode-capable language bindings. Such differences are mentioned in the corresponding API descriptions in this chapter.

The *PLOP_convert_to_unicode( )* function can be used to convert between UTF-8, UTF-16, and UTF-32 strings or from arbitrary encodings to Unicode with an optional BOM.

8.1.3 Multi-threaded Programming

While PLOP itself is single-threaded, it can safely be used in multi-threaded applications. In the common situation that a PLOP object is only used within one thread, no particular multi-threading precautions are necessary. If the same object is used in multiple threads the application must synchronize the threads to make sure that the PLOP object is not accessed simultaneously by more than one thread. A typical scenario would involve a pool of PLOP objects where each thread fetches an existing PLOP object from the pool instead of creating a new one, and returns it to the pool after creating a document if the object is no longer needed. Using the same PLOP object in another thread before the output document is finished will rarely provide any advantage and is not recommended.

8.2 General Functions

C ***PLOP *PLOP_new(void)***

Create a new PLOP context.

Returns A handle to the new context, or NULL if not enough memory is available. The context must be supplied to all other API methods.

Bindings Not available in object-oriented language bindings where it will be called automatically when a new PLOP object is created.

Java ***void delete( )***
C# ***void Dispose( )***
C ***void PLOP_delete(PLOP *plop)***

Delete a PLOP context and release all its internal resources.

Details All open documents in the context are closed automatically. It is good programming practice, however, to close documents explicitly with *close_document( )* when they are no longer needed.

Bindings In C this function must not be called within a *PLOP_TRY( )/PLOP_CATCH( )* clause.

In Java this method will be called by the finalizer method of PLOP. However, it is strongly recommended to explicitly call *delete( )* for reliable cleanup. The same holds true when an exception occurred.

In Perl and PHP this function will be called automatically when the PLOP object is destroyed.

In .NET *Dispose( )* should be called at the end of processing to clean up unmanaged resources.

C++ ***void create_pvf(wstring filename, const void *data, size_t size, wstring optlist)***
C# Java ***void create_pvf(String filename, byte[ ] data, String optlist)***
Perl PHP ***create_pvf(string filename, string data, string optlist)***
C ***void PLOP_create_pvf(PLOP *plop,***
const char *filename, int len, const void *data, size_t size, const char *optlist)

Create a named virtual read-only file from data provided in memory.

filename (Name string) The name of the virtual file. This is an arbitrary string which can later be used to refer to the virtual file in other PLOP calls.

len (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* a null-terminated string must be provided.

data Data for the virtual file. In C and C++ this is a pointer to a memory location. In Java this is a byte array. In Perl and PHP this is a string.

size (C and C++ only) The length in bytes of the memory block containing the data.

optlist An option list according to Table 8.1. The following option can be used: *copy*.

Details This function may be useful for repeatedly used digital IDs or XMP metadata. The virtual file name can be supplied to any API method which uses input files. Some of these functions may set a lock on the virtual file until the data is no longer needed. Virtual files will be kept in memory until they are deleted explicitly with *delete_pvf( )*, or automatically in *delete( )*.

Each PLOP object maintains its own set of PVF files. Virtual files cannot be shared among different PLOP objects. Multiple threads working with separate PLOP objects do not need to synchronize PVF use. If *filename* refers to an existing virtual file an exception will be thrown. This function does not check whether *filename* is already in use for a regular disk file.

Unless the *copy* option has been supplied, the caller must not modify or free (delete) the supplied data before a corresponding successful call to *delete_pvf( )*. Not obeying to this rule will most likely result in a crash.

Table 8.1 Option for create_pvf( )

option	description
copy	*(Boolean) If* `true`*, PLOP creates an internal copy of the supplied data. In this case the caller may dispose of the supplied data immediately after this call. Default:* `false` *for C and C++, but* `true` *for all other language bindings*

C++ Java C# ***int delete_pvf(String filename)***

Perl PHP ***int delete_pvf(string filename)***

C ***int PLOP_delete_pvf(PLOP *plop, const char *filename, int len)***

Delete a named virtual file and free its data structures.

filename (Name string) The name of the virtual file as supplied to *create_pvf( )*.

len (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* a null-terminated string must be provided.

Returns -1 (in PHP: 0) if the corresponding virtual file exists but is locked, and 1 otherwise.

Details If the file isn't locked, PLOP will immediately delete the data structures associated with *filename*. If *filename* does not refer to a valid virtual file this function will silently do nothing. After successfully calling this function *filename* may be reused. All virtual files will automatically be deleted in *delete( )*.

The detailed semantics depend on whether or not the *copy* option has been supplied to the corresponding call to *create_pvf( )*: If the *copy* option has been supplied, both the administrative data structures for the file and the actual file contents (data) will be freed; otherwise, the contents will not be freed, since the client is supposed to do so.

C++ Java C# **double info_pvf(String filename, String keyword)**

Perl PHP **float info_pvf(string filename, string keyword)**

C **double PLOP_info_pvf(PDF *p, const char *filename, int len, const char *keyword)**

Query properties of a virtual file or the PDFlib Virtual File system (PVF).

filename (Name string) The name of the virtual file. The filename may be empty if *keyword=filecount*.

len (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* a null-terminated string must be provided.

keyword A keyword according to Table 8.2.

Table 8.2 Keywords for info_pvf()

keyword	description
exists	*1 if the file exists in the PDFlib Virtual File system (and has not been deleted), otherwise 0*
filecount	*Total number of files in the PDFlib Virtual File system maintained for the current PLOP object. The* `filename` *parameter will be ignored.*
iscopy	*(Only for existing virtual files) 1 if the* copy *option was supplied when the specified virtual file was created, otherwise 0*
lockcount	*(Only for existing virtual files) Number of locks for the specified virtual file set internally be PLOP functions. The file can only be deleted if the lock count is 0.*
size	*(Only for existing virtual files) Size of the specified virtual file in bytes.*

Returns The value of some file parameter as requested by *keyword*.

Details This function returns various properties of a virtual file or the PDFlib Virtual File system (PVF). The property is specified by *keyword*.

8.3 Input Functions

C++ Java C# ***int open_document(String filename, String optlist)***

Perl PHP ***int open_document(string filename, string optlist)***

C ***int PLOP_open_document(PLOP *plop, const char *filename, int len, const char *optlist)***

Open a PDF document (which may be protected) for processing.

filename The full path name of the PDF file to be opened. The file will be searched by means of the *SearchPath* resource.

In non-Unicode language bindings the file name is converted to UTF-8 according to the *filenamehandling* option (unless *filenamehandling=unicode* or the supplied file name starts with a UTF-8 BOM). If *len* is different from 0 (C language binding only) the file name is converted from UTF-16 to UTF-8 regardless of the option *filenamehandling*. An error occurs if the file name cannot be converted or if the file name does not constitute valid UTF-8 or UTF-16.

On Windows it is OK to use UNC paths or mapped network drives as long as you have the necessary permissions (which may not be the case when running in ASP).

len (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* a null-terminated string must be provided.

optlist An option list (see Section 8.1, »Option Lists and other General Topics«, page 135) according to Table 8.3.

Returns -1 (in PHP: 0) on error, and a document handle otherwise. After an error it is recommended to call *get_errmsg( )* to find out more details about the error.

Details The document handle can be used for the following purposes:

- use as input document for further processing with *create_document( );*
- provide a page as signature appearance (signature option *field* and suboption *visdoc);*
- query document information with pCOS.

If the document is encrypted its user or master password (or a suitable digital ID in case of certificate security) must be supplied unless the *requiredmode* option has been specified.

Table 8.3 Options for open_document( ) and open_document_callback( )

option	description
digitalid	*(Option list; only for input documents which are protected with certificate security) Digital ID of a recipient for which the document is encrypted. The digital ID is specified with the suboptions in Table 8.8. If* `engine=mscapi` *the* `digitalid` *option may be missing or empty which means that all available IDs in the default certificate store are tried.*
engine	*(Keyword; only for input documents which are protected with certificate security) Cryptographic engine for decrypting the document (default:* `builtin`*):* ***builtin*** *Use the built-in cryptographic engine; the* `digitalid` *option must specify a virtual or disk file.* ***mscapi*** *(Only on Windows) Use Microsoft Crypto API as cryptographic engine. The digital ID can be supplied in the certificate store or a disk file. The* `digitalid` *option may be missing or empty.*

Table 8.3 Options for open_document( ) and open_document_callback( )

option	description
inmemory	*(Boolean; only for open_document( )) If* `true`*, PLOP loads the complete file into memory and process it from there. This can result in a tremendous performance gain on some systems (especially MVS) at the expense of memory usage. If* `false`*, individual parts of the document will be read from disk as needed. Default:* `false`
password	*(String; required for protected documents except with* `requiredmode`*)* *For documents protected with password security: user or master password for the document. As detailed in Table 5.2, page 65, the document's user password, master password, or no password may be required depending on which operation is applied to the document. On EBCDIC platforms the password is expected in* `ebcdic` *encoding or EBCDIC-UTF-8. If* `update=true` *the same password is used as master password for the generated output document.* *If* `digitalid` *is supplied: Password, pass phrase, or PIN for the digital ID required for documents protected with certificate security. For* `engine=builtin` *exactly one of* `password` *or* `passwordfile` *is required; other engines may use alternate methods. On EBCDIC platforms the password is expected in* `ebcdic` *encoding.*
passwordfile	*(String; for* `engine=builtin` *exactly one of* `password` *or* `passwordfile` *is required; other engines may use alternate methods) The first line of the file (excluding the line end character or characters) is used as password, pass phrase, or PIN for the digital ID. On EBCDIC platforms the contents of the password file are expected in* `ebcdic` *encoding.*
pcosengines	*(Option list) Enable or disable pCOS engines for document-wide resource collection. Turning off an engine means that the corresponding per-document and per-page pseudo object arrays will be empty. Supported keywords (default: all pCOS engines are active):* `colorspace`, `extgstate`, `font`, `image`, `pattern`, `property`, `shading`, `template`
repair	*(Keyword) Specifies how to treat damaged PDF input documents. Repairing a document takes more time than normal parsing, but may allow processing of certain damaged PDFs. Note that some documents may be damaged beyond repair (default:* `auto`*):* ***force*** *Unconditionally try to repair the document, regardless of whether or not it has problems.* ***auto*** *Repair the document only if problems are detected while opening the PDF.* ***none*** *No attempt will be made at repairing the document. If there are problems in the PDF the function call will fail.*
requiredmode	*(Keyword) The minimum pcos mode* (`minimum`/`restricted`/`full`) *which is acceptable when opening the document. The call fails if the resulting pCOS mode would be lower than the required mode. If the call succeeds it is guaranteed that the resulting pCOS mode is at least the one specified in this option. However, it may be higher; e.g.* `requiredmode=minimum` *for an unencrypted document results in* `full` *mode. Default:* `full`
shrug	*(Boolean) Permission restrictions are ignored (i.e. PDF processing is allowed) if the document could only be opened in restricted pCOS mode otherwise. For password security this means that the document is encrypted with a master password, but only the user password (if any) has been supplied. For certificate security this means that a suitable recipient digital ID has been supplied, but the document does not set master permission for this ID. If permission restrictions are ignored, the pCOS pseudo object* `shrug` *is set to* `true`*. Default:* `false`

Table 8.3 Options for open_document( ) and open_document_callback( )

option	description
xmppolicy	*(Keyword) Control treatment of invalid document-level XMP in the input document. Invalid XMP implies that no standard identifier can be found, e.g. PDF/A documents will not be treated as such. Supported keywords (default:* `rejectinvalid`*):*
	rejectinvalid *Throw an exception for invalid XMP which includes the XML parser error message, and stop processing.*
	ignoreinvalid *(Implies* `sacrifice={pdfa pdfua pdfvt pdfx}`*) Treat invalid XMP as if there was no XMP present. Output XMP will be generated based on document info entries; it will also include an XML parsing error message in the* `<pdfx:Exception>` *element.*
	remove *(Implies* `sacrifice={pdfa pdfua pdfvt pdfx}`*) Unconditionally ignore input XMP, regardless of its validity. The output XMP is generated from scratch. This may be useful to delete unwanted metadata. Standard identifiers in XMP (e.g. for PDF/A) are lost.*

C++ ***int open_document_callback(void *opaque, plop_off_t filesize,***
size_t (*readproc)(void *opaque, void *buffer, size_t size),
int (*seekproc)(void *opaque, plop_off_t offset),
wstring optlist)

C ***int PLOP_open_document_callback(PLOP *plop, void *opaque, plop_off_t filesize,***
size_t (*readproc)(void *opaque, void *buffer, size_t size),
int (*seekproc)(void *opaque, plop_off_t offset),
const char *optlist)

Open a PDF document (which may be protected) via a user-supplied function.

opaque Pointer to some opaque data structure which will be passed to *readproc*. PLOP does not use this pointer or the underlying data.

filesize The length of the document in bytes.

readproc A procedure which must be able to supply arbitrary chunks of *size* bytes of the document at memory location *buffer*. The procedure must return the number of bytes retrieved.

seekproc A procedure for seeking to position *offset* within the document. The procedure must return -1 in case of error, and 0 otherwise.

optlist An option list (see Section 8.1, »Option Lists and other General Topics«, page 135) according to Table 8.3.

Returns -1 (in PHP: 0) on error, and a document handle otherwise. After an error it is recommended to call *get_errmsg( )* to find out more details about the error.

Details See *open_document( )*. The callback functions must not call any API method or macro with the same context pointer.

Bindings Only available in the C and C++ language bindings. The *plop_off_t* type is defined conditionally in *ploplib.h*. It usually holds 64-bit values as offset type for large files beyond 2GB. The application must be built with Large File Support (LFS).

C++ Java C# ***void close_document(int doc, String optlist)***

Perl PHP ***close_document(long doc, string optlist)***

C ***void PLOP_close_document(PLOP *plop, int doc, const char *optlist)***

Close the specified document.

doc A valid document handle obtained with *open_document*()*.

optlist An option list (currently unused).

Details This function should be called before *delete()* for each document which has been opened with *open_document*()*. It closes the document associated with the supplied handle and releases all related resources.

8.4 Output Functions

C++ Java C# ***int create_document(String filename, String optlist)***

Perl PHP ***int create_document(string filename, string optlist)***

C ***int PLOP_create_document(PLOP *plop, const char *filename, int len, const char *optlist)***

Create a PDF output document in memory or on disk file.

filename (Name string) The name of the generated output file, which must be different from the input file name supplied to *open_document( )*. If this is an empty string the output is generated in memory, and can later be fetched with *get_buffer( )*.

In non-Unicode language bindings file names with *len=0* are interpreted in the current system codepage unless they are preceded by a UTF-8 BOM, in which case they are interpreted as UTF-8 or EBCDIC-UTF-8.

len (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* a null-terminated string must be provided.

optlist An option list (see Section 8.1, »Option Lists and other General Topics«, page 135) according to Table 8.4.

Returns -1 (in PHP: 0) on error, and a document handle otherwise. After an error it is recommended to call *get_errmsg( )* to find out more details about the error.

When a digital signature is created the function call fails in the following cases:

- a timestamp could not be obtained and the *critical* option is set;
- the signature chain is revalidated and a certificate has expired or has been revoked in the meantime;
- a visualization document is supplied which doesn't fit on the page;
- the input document is damaged and the signature is created in update mode.

If *add_recipient( )* has been called one or more times, but none of the calls was successful (i.e. all recipient certificates have been rejected), *create_document( )* fails. This is to avoid accidental creation of unprotected output.

Details Before calling this function *open_document*( )* must have been called. The document to be processed is supplied in the *input* option. See Section 5.2, »Password-protecting PDF Documents with PLOP«, page 65, for conditions which are enforced for the user and master passwords.

In signature mode create_document( ) may revalidate the signature chain, e.g. because an OCSP response expired since it has been requested.

In certificate security mode, i.e. if *add_recipient( )* has been called at least once with a non-empty *certificate* option list, this function checks whether any recipient certificate has expired. If so, the function call fails.

Table 8.4 Options for create_document( )

option	description
docinfo	*(List of pairs of text strings) Set document info entries for the output document. If the document contains document XMP metadata, standard document info entries are mirrored in the XMP. Each pair in the option list contains the name of an entry and its value. The supplied info entries are always synchronized to XMP, even if no document info dictionary is emitted for PDF 2.0 (see option* `emitdocinfo`*). The following predefined and custom keys can be supplied (default: document info entries are copied from the input document):* ***Subject*** *Subject of the document* ***Title*** *Title of the document* ***Author*** *Author of the document* ***Keywords*** *Keywords describing the contents of the document* ***Trapped*** *Indicates whether trapping has been applied to the document. Allowed values are* `True`*,* `False`*, and* `Unknown`*. For PDF/X input* `Unknown` *is only allowed if the* `sacrifice` *option includes* `pdfx` *or* `pdfvt`*.* ***any name other than Creator, CreationDate, Producer, ModDate, GTS_PDFXVersion, GTS_PDFXConformance, ISO_PDFEVersion*** *User-defined field name. PLOP supports an arbitrary number of fields. A custom field name should be supplied only once.*
emitdocinfo	*(Boolean; only relevant for PDF 2.0 output) If* `true`*, a document information dictionary is emitted. Default:* `false`
encryption	*(Keyword; only relevant for password security and certificate security) Encryption algorithm to be used for protecting the output document.* *The following keywords are supported for password security, i.e. if* `masterpassword` *is supplied (default:* `algo11`*):* ***algo4*** *(Deprecated in PDF 2.0) Encrypt with AES-128 according to Acrobat 7/8, i.e. pCOS algorithm 4; this increases the output PDF version to PDF 1.6 if required. Passwords may contain only Latin-1 characters and are truncated to 32 characters.* ***algo11*** *Encrypt with AES-256 according to Acrobat X/XI/DC, i.e. pCOS algorithm 11; this increases the output PDF version to PDF 1.7ext8 if required. Passwords may contain Unicode characters and are truncated to 127 UTF-8 bytes.* *The following keywords are supported for certificate security, i.e. add_recipient( ) has been called at least once successfully with a non-empty* `certificate` *option list (default:* `algo10`*):* ***algo6*** *(Deprecated in PDF 2.0) Encrypt with certificate security on top of AES-128 according to Acrobat 7, i.e. pCOS algorithm 6; this increases the output PDF version to PDF 1.6 if required.* ***algo10*** *Encrypt with certificate security on top of AES-256 according to Acrobat 9, i.e. pCOS algorithm 10; this increases the output PDF version to PDF 1.7ext3 if required.*
input	*(Document handle obtained with open_document*( ); required) Input document to be processed*
limitcheck	*If* `true`*, the limit for the number of indirect PDF objects (8.388.607) is enforced in PDF/A-1/2/3 and PDF/X-4/5 modes. Default:* `true`
linearize	*(Boolean; cannot be combined with signature creation or* `metadata`*) If* `true`*, the output document will be linearized. On MVS systems this option cannot be combined with in-memory generation (i.e. empty* `filename` *parameter). Default:* `false`
master-password[1]	*(String; forces* `update=false`*) Master password for password-protecting the document. If it is empty or missing no master password is applied. Default: empty*

Table 8.4 Options for create_document( )

option	description
metadata	*(Option list; can not be combined with* `linearize`*) Supply XMP metadata for the document. PDF/A and PDF/X conformance entries are not allowed in the supplied XMP. Supported suboptions:* ***filename*** *(Name string; required) The name of a file containing well-formed XMP metadata in UTF-8 format.* ***validate*** *(Keyword) The supplied XMP metadata will be validated according to the keyword (note that PLOP does not validate the XMP metadata in the input document):* ***none*** *No validation* ***xmp2004*** *Validation according to the XMP 2004 specification* ***xmp2005*** *Validation according to the XMP 2005 specification* ***pdfa1*** *Like* `xmp2004`*, plus testing for predefined properties and schemas, and extension schema validation according to PDF/A-1* ***pdfa2*** *Like* `xmp2005`*, plus testing for predefined properties and schemas, and extension schema validation according to PDF/A-2 and PDF/A-3 (both standards have identical metadata requirements)* *Default:* `pdfa1` *if the input conforms to PDF/A-1 and the* `sacrifice` *option does not include* `pdfa`*;* `pdfa2` *if the input conforms to PDF/A-2 or PDF/A-3 and the* `sacrifice` *option does not include* `pdfa`*; otherwise* `none`
objectstreams	*(Keyword; forced to* `none` *for* `linearize=true` *as well as in PDF/A-1 and PDF/X-1a/3 modes) Create compressed object streams which significantly reduce output file size (default:* `all`*):* ***all*** *Write all simple objects except the document info dictionary into compressed objects streams and create a compressed cross-reference stream.* ***none*** *Don't create any compressed object streams nor compressed cross-reference stream.* ***xref*** *Generate a compressed cross-reference stream, but not any other compressed object streams.*
optimize	*(Keyword; ignored if* `update=true`*) Optimizations to be applied (default:* `none`*):* ***all*** *Apply resource optimizations.* ***none*** *Don't apply any optimization.*
permissions	*(Keyword list; requires* `masterpassword` *or certificate security; not allowed if* `update=true`*) List of permission restrictions for the document. It contains any number of the keywords* `noprint`*,* `nomodify`*,* `nocopy`*,* `noannots`*,* `noassemble`*,* `noforms`*,* `noaccessible`*,* `nohiresprint`*, and* `plainmetadata` *(see Table 5.3, page 66).* *In certificate security mode only the keyword* `plainmetadata` *is allowed; other permission restrictions can be specified in the* `permissions` *option of add_recipient( ).* *Default: empty*
recordsize	*(Integer; z/OS and USS only) The record size of the output file. Default: 0 (unblocked output)*
sacrifice	*(List of keywords) This option can be used for controlling the behavior in case of conflicts between properties of the input PDF and the requested action. By default, PLOP does not create any output if it detects a conflict, but throws an exception instead. However, you can sacrifice some property of the document in order to allow processing. The keywords listed in Table 8.5. are supported; they are ignored unless both the input and action triggers are true. Default: empty list, i.e. an exception is thrown in case of a conflict and no output will be created.*
tempdirname	*(String) Name of a directory where temporary files needed for PLOP's internal processing will be created. If empty, PLOP will generate temporary files in the current directory. This option will be ignored if the* `tempfilename` *option has been supplied. Default: empty*
tempfilename	*(String; MVS only) Full file name for a temporary file needed for PLOP's internal processing. If empty, PLOP will generate a unique temp file name. The user is responsible for deleting the temporary file after close_document( ). If this option is supplied the* `filename` *parameter must not be empty. Default: empty*
user-password[1]	*(String; requires* `masterpassword`*) User password for password-protecting the document. If it is empty or missing no user password is applied. Default: empty*

1. Arbitrary Unicode characters can be supplied for AES-256 (algorithm 11), but only Latin-1 characters for AES-128 (algorithm 4). The supplied password is truncated to 127 UTF-8 bytes for algorithm 11 and to 32 characters for algorithm 4. On EBCDIC platforms the password must be supplied in ebcdic *encoding or EBCDIC-UTF-8.*

Table 8.5 Keywords for the sacrifice *option of create_document( )*

keyword	*description*
encrypted-attachments	*(Input trigger: the document is not encrypted, but contains one ore more encrypted file attachments; action trigger: the appropriate password for the encrypted file attachment has not been supplied with the* password *option). If this keyword is supplied, encrypted file attachments for which the password is not available are removed.* *Documents containing encrypted file attachments for which the proper password has not been supplied cannot be processed at all when signing with* update=true*.*
fields	*(Input trigger: the document contains one or more non-signature form fields with* NeedAppearances=true*; action trigger: signing with* update=false*). If this keyword is supplied, all form fields including signature fields are removed.*
pdfa	*(Input trigger: the document conforms to any conformance level of PDF/A-1, PDF/A-2 or PDF/A-3; action triggers: signature creation with option* visdoc *and an incompatible visualization document, any of the options* userpassword/masterpassword/permissions*, certificate security mode, or a signature gets larger than 64K for PDF/A-1 and 32K for PDF/A-2/3) If this keyword is supplied, PDF/A input can be processed, but the PDF/A conformance entries are removed.*
pdfua	*(Input trigger: the document conforms to PDF/UA-1; action triggers: encryption with the option* permissions *and the keyword* noaccessible*) If this keyword is supplied, output can be created which no longer conforms to PDF/UA and the PDF/UA conformance entries are removed.*
pdfvt	*(Input trigger: the document conforms to PDF/VT-1; action triggers: same as for* pdfx*) If this keyword is supplied, PDF/VT input can be processed, but the PDF/VT and PDF/X conformance entries are removed.*
pdfx	*(Input trigger: the document conforms to PDF/X-1a or PDF/X-3/4/5; action triggers: signature creation in combination with* Trapped=Unknown *in the* docinfo *option, or with the* visdoc *suboption of the* field *option, or any of the options* userpassword/masterpassword/permissions*, or certificate security mode) If this keyword is supplied, PDF/X input can be processed, but PDF/X conformance entries are removed. If the document also conforms to PDF/VT-1 the PDF/VT conformance entries are removed as well.*
signatures	*(Input trigger: the document contains one or more signatures; action trigger: all actions except signing with* update=true*; if the document contains a certification signature which doesn't allow changes, signing with* update=true *is also a trigger*). *If this keyword is supplied and both input and action triggers are true, existing signatures are cleared (i.e. signature values, but not the corresponding form fields are removed) in order to avoid creating output with invalid signatures. If both input and action triggers are true and* sacrifice={signatures} *is not supplied, create_document( ) fails.*

C++ ***const char *get_buffer(long *size)***
C# Java ***byte[ ] get_buffer( )***
Perl PHP ***string get_buffer( )***
C ***const char *PLOP_get_buffer(PLOP *plop, long *size)***

Fetch full or partial buffer contents of the output document from memory.

size Only required in the C binding. A pointer to a memory location where the length of the returned buffer will be stored.

Returns A buffer containing output data. The client must consume the buffer contents before calling any other PLOP library function.

Details PDF output can only be fetched with this function if in-memory generation has been requested by supplying an empty file name to *create_document( )* (otherwise output will be written to a file). *get_buffer( )* must be called before calling *close_document( )*.

8.5 Certificate Security

C++ Java C# ***int add_recipient(String optlist)***

Perl PHP ***int add_recipient(string optlist)***

C ***int PLOP_add_recipient(PLOP *plop, const char *optlist)***

Add a recipient certificate for protecting the output document.

optlist Option list (see Section 8.1, »Option Lists and other General Topics«, page 135) specifying recipient information according to Table 8.6:
certificate, *conformance*, *engine*, *oaephash*, *rsapadding*, *permissions*

Returns -1 (in PHP: 0) on error, and 1 otherwise. After an error it is recommended to call *get_errmsg( )* to find out more details about the error. The function call fails if the certificate cannot be found or cannot be used for encrypting PDF documents.

Details If this function is called at least once with a non-empty *certificate* option the output document(s) is encrypted against all specified recipient certificate(s). Certificates can be supplied in a disk-based or virtual file or in the Windows certificate store. This function can be called an arbitrary number of times to build the list of recipients for the document. For most use cases it is recommended to include the document author's certificate in the recipient list since otherwise the author will be unable to open the protected document.

The supplied certificate must match the conditions described in »Requirements for recipient certificates«, page 80.

An arbitrary number of documents can be encrypted against the same recipient list with multiple calls to *create_document( )*. However, if *add_recipient( )* is called again after *create_document( )* it first resets the recipient list to an empty list so that a new list of recipients can be created.

This function forces the signature suboption value *update=false* since the recipient list cannot be modified in update mode.

Table 8.6 Options for add_recipient( )

option	*description*
certificate	*(Option list; required) Specify a recipient certificate against which the document will be encrypted. The certificate is specified with suboptions according to Table 8.8. For security reasons the file is not searched in the* `searchpath`. *An empty lists disables certificate security. This may be useful to switch between certificate-based encryption and other processing.*
conformance	*(Keyword) Conformance of the generated encryption information (default:* `acrobat`*):* ***acrobat*** *The document can be opened with Acrobat DC. If the recipient certificate uses an algorithm which is not supported in Acrobat the call fails.* ***extended*** *Accept the recipient certificate even if it uses one of the features below. It may not be possible to open the document with Acrobat:* *RSA certificates where the key length is not a multiple of 8;* *ECC certificates with a Brainpool curve (RFC 5639).* *Encryption with option* `rsapadding=oaep`.

Table 8.6 Options for add_recipient( )

option	*description*
engine	*(Keyword) Cryptographic engine to be used for locating the recipient's certificate (default:* `builtin`*):* ***builtin*** *Use the built-in cryptographic engine; certificates must be supplied in a virtual or disk file.* ***mscapi*** *(Only on Windows) Use Microsoft Crypto API as cryptographic engine; certificates can be supplied in the certificate store or a disk file.*
oaephash	*(Keyword; only relevant for* `rsapadding=oaep`*) Hash function for use in OAEP (default:* `sha256`*):* `sha1`*,* `sha256`*,* `sha384`*, or* `sha512`
rsapadding	*(Keyword) Padding mechanism for RSA key transport (default: pkcs#1):* ***pkcs#1*** *RSA padding according to PKCS#1 v1.5. This scheme is supported in all Acrobat versions.* ***oaep*** *OAEP (Optimal asymmetric encryption padding) according to RFC 3447. This method offers security advantages. Since OAEP is not supported in Acrobat DC this keyword requires the option* `conformance=extended`*.*
permissions	*(Keyword list) List of permission restrictions for the recipient. Multiple recipients may be assigned different permission restrictions. The list contains any number of the permission restriction keywords* `noprint`*,* `nomodify`*,* `nocopy`*,* `noannots`*,* `noassemble`*,* `noforms`*,* `noaccessible`*,* `nohiresprint` *according to Table 5.3. The following additional keyword can be used:* ***nomaster*** *Restrict printing, editing, and content extraction according to the specified permission restriction keywords, and prevent changing the document's security settings. If neither this keyword nor one of the other permission restriction keywords above is supplied, the recipient has full rights to the document. Setting any of the other permission restriction keywords above implies* `nomaster`*.* *The keyword* `plainmetadata` *can not be used here, but must be supplied to create_document( ) as it doesn't apply to individual recipients.* *An empty list means that no permission restrictions apply to the recipient. Default: empty list*

8.6 Digital Signatures

Note *Digital signature functionality is only available in the product PLOP DS.*

C++ Java C# ***int prepare_signature(String optlist)***

Perl PHP ***int prepare_signature(string optlist)***

C ***int PLOP_prepare_signature(PLOP *plop, const char *optlist)***

Prepare signature options.

optlist Option list specifying signature options according to Table 8.7:

- Options for the signing certificate (digital ID): *digitalid, password, passwordfile*
- Options for providing information about the signature context: *contactinfo, location, policy, reason*
- Options for timestamping: *doctimestamp, timestamp*
- Option for signature visualization: *field*
- Options for providing validation information: *certfile, crl, crldir, crlfile, ocsp, rootcertdir, rootcertfile, validate*
- Options for certification signatures: *certification*, *preventchanges*
- Options for controlling details of signature creation: *conformance, engine, ltv, signature, sigtype*
- Options for controlling signature details: *dss, rsaencoding, timestampsize, update*

Returns -1 (in PHP: 0) on error, and 1 otherwise. After an error it is recommended to call *get_errmsg( )* to find out more details about the error. The function call may fail for the following reasons:

- the signer's digital ID cannot be found or the private key cannot be accessed, e.g. because of a wrong password or PIN;
- validation fails, e.g. no valid OCSP response or CRL could be retrieved and the corresponding *critical* option is set;
- the requirements for *ltv=full* or *validate=full* cannot be fulfilled.

After a failed call to *prepare_ signature( )* it is recommended to avoid another call with the same options since this may disable a PKCS#11 token, e.g. if a wrong password/PIN was supplied too often.

Details Signature options prepared with this function can be used to create an arbitrary number of signatures with *create_document( )*. The supplied signature options will be used for all signatures created with *create_document( )* until *prepare_ signature( )* is called again (with other signature options or the option *nosignature)*.

The signature preparation option list may be processed again at unpredictable times before a signature is created. In particular, if a CRL or OCSP response is found to be outdated after a number of signatures has been created, the signature options are processed again to refresh certificate revocation information.

This function terminates a PKCS#11 session which may be active from a previous call. If you need to terminate a PKCS#11 session explicitly (e.g. to provide other threads access to the token) you can call this function with the option *signature=false*.

Table 8.7 Options for prepare_signature( )

option	*description*
certfile	*(String; not for* `engine=mscapi`*) Name of a file which contains one or more intermediate CA certificates in PEM encoding which may be required for validating and embedding the full certificate chain.*
certification	*(Keyword) Create a certification (author) signature with the specified certification level. Values other than* none *create a certification signature and should only be used for the first signature in a document (default:* `none`*):* ***formfilling*** *Certification signature: form filling and signing (by clicking a signature field, but not via Acrobat's menu items) are allowed. Adding pages by spawning page templates is also allowed (as opposed to manually adding pages), but this technique is rarely used. Other changes break the signature.* ***formsandannotations*** *Certification signature: form filling, signing and page adding as well as commenting (i.e. annotation creation, deletion, and modification) are allowed; other changes break the signature.* ***nochanges*** *Certification signature: any change breaks the signature.* ***none*** *Regular approval signature: document is not certified.*
conformance	*(Keyword) Conformance of the generated signature (default:* `acrobat`*):* ***acrobat*** *The signature can be validated with Acrobat DC. The call fails if a certificate or options are used which are not Acrobat-compatible.* ***extended*** *Accept the signing certificate and options even if the resulting signature cannot be validated with Acrobat DC. The following features require* `conformance=extended`*:* *ECC certificates with one of the Brainpool curves (RFC 5639)* *Signature option* `ocsp` *where the suboption* `hash` *has a value different from* `sha1`*.*
contactinfo	*(Text string; only relevant for* `digitalid`*) Information provided by the signer to enable a recipient to contact the signer to verify the signature (e.g. a phone number). However, this is not recommended as a scalable solution for establishing trust. Acrobat 8/9/X display the contact information in the* `Signature Properties` *dialog in the* `Signer` *tab. Acrobat XI/DC does not display the contact information.*
crl	*(Option list or keyword; except for* `crl=none` *only relevant for* `digitalid`*; not for* `engine=mscapi`*) Obtain a certificate revocation list (CRL) for the signing certificate and embed it in the signature or DSS if no valid good OCSP response is available. Supported suboptions (default:* `{source={ } critical=false}`*, i.e. the* `CRLdp` *extension in the digital ID is used if present):* ***critical*** *(Boolean) If* `true`*, a signature is only generated if a valid CRL could be retrieved for the signing certificate; otherwise an error is returned and no signature is created. If this option is* `false` *CRL embedding is silently ignored if no valid CRL could be retrieved. Default:* `true` ***filename*** *(String) Name of a file containing a CRL for the signing certificate in DER encoding. If the* `filename` *option is present a* `CRLdp` *extension in the signing certificate is ignored.* ***source*** *(Network option list) Option list describing the CRL distribution point for the signing certificate. The protocols* `http` *and* `https` *are supported. The* `url` *suboption of the* `source` *option or the* `source` *option itself can be omitted which means that the CRL distribution point* (`CRLdp`) *extension in the digital ID is used as source.* *All suboptions of the* `source` *network option list except* `url` *and* `httpauthentication` *are also applied to CRL requests for certificates other than the signing certificate. This supports use of the same set of credentials (e.g.* `username`/`password`) *in CRL calls for all involved certificates.* *Unless a* `CRLdp` *extension is present in the digital ID exactly one of the* `filename` *or* `source` *options must be supplied.* *The option* `crl=none` *means that no CRLs are retrieved over the network even if a* `CRLdp` *extension is present. This affects all involved certificates, not just the signing certificate.*
crldir	*(String; not for* `engine=mscapi`*) Name of a directory containing CRLs in PEM encoding which may be required for validating the involved certificates. See »Naming convention for certificate and CRL files«, page 175, regarding the file names.*

Table 8.7 Options for prepare_signature( )

option	description
crlfile	*(String; not for* `engine=mscapi`*) Name of a file containing one or more CRLs in PEM encoding which may be required for validating the involved certificates.*
digitalid	*(Option list; required for approval and certification signatures) Specify the signer's digital ID with suboptions according to Table 8.8. The supported suboptions depend on the selected engine.*
doc-timestamp	*(Option list; not for* `engine=mscapi`*) Generate a document-level timestamp from a trusted timestamp authority (using the* `builtin` *engine). Supported suboptions: see option timestamp*
dss	*(Boolean; not for* `engine=mscapi`*) If* `true`*, embed certificates and revocation information in a Document Security Store (DSS) (see Section 7.3.3, »Document Security Store (DSS)«, page 106). Otherwise this data is embedded in the signature. Validation information for embedded timestamps and document timestamps is always embedded in a DSS regardless of this option. Default:* `true` *for* `sigtype=cades` *as well as for input documents with an existing DSS;* `false` *otherwise*
engine	*(Keyword) Cryptographic engine to be used for signing (default:* `builtin`*):* **builtin** *Use the built-in cryptographic engine; the digital ID must be supplied in a virtual or disk file.* **mscapi** *(Only on Windows) Use Microsoft Crypto API as cryptographic engine; the digital ID can be supplied in the certificate store or a disk file.* **pkcs#11** *Use the PKCS#11 interface to load the digital ID from a cryptographic token. The name of the corresponding PKCS#11 DLL/shared library for the token must be provided in the* `filename` *suboption of the* `digitalid` *option.*
field	*(Option list; only relevant for* `digitalid`*) Coordinates and contents of the form field which holds the signature according to the suboptions in Table 8.9. Default: an invisible signature is created*
location	*(Text string; only relevant for* `digitalid`*) Physical location or host name where the signature is created*
ltv	*(Keyword; not for* `engine=mscapi`*) Specify whether the signed document is prepared for long-term validation (LTV) (default:* `try`*):* **full** *(Implies* `validate=full`*) Embed full validation information to LTV-enable the signed document. LTV status usually requires one of the* `rootcertdir` *or* `rootcertfile` *options; the options* `certfile`*,* `ocsp` *and* `crl` *for providing additional certificates and revocation information may also be required. The call fails if a required certificate or revocation information for a certificate cannot be obtained.* **none** *Don't embed validation information. The signed document is smaller, but not LTV-enabled.* **try** *Embed as much validation information as is available. The signed document may or may not be LTV-enabled depending on available certificates and revocation information.*
ocsp	*(Option list or keyword; not for* `engine=mscapi`*) Configure OCSP handling with suboptions according to Table 8.10. Default:* `{source={ } critical=false}`*, i.e. the AIA extension in the digital ID is used if present.* *The option* `ocsp=none` *means that no OCSP responses are retrieved even if an AIA extension is present. This affects all involved certificates, not just the signing certificate.*
password	*(String which may be empty; for* `engine=builtin` *exactly one of* `password` *or* `passwordfile` *is required; other engines may use alternate methods) Specifies the password, pass phrase, or PIN for the digital ID. For* `engine=pkcs#11` *this option must contain the PIN for the cryptographic token unless the PIN must be entered interactively on the token itself (e.g. a smartcard reader with keyboard). On EBCDIC platforms the password is expected in* `ebcdic` *encoding.*
passwordfile	*(String; for* `engine=builtin` *exactly one of* `password` *or* `passwordfile` *is required; other engines may use alternate methods) The first line of the file (excluding the line end character or characters) is used as password, pass phrase, or PIN for the digital ID. On EBCDIC platforms the contents of the password file are expected in* `ebcdic` *encoding.*

Table 8.7 Options for prepare_signature( )

option	description
policy	*(Option list; only for* `sigtype=cades`*; not allowed if* `reason` *is specified; required for PAdES E-EPES) Signature policy which shall be used to validate the signature. Supported suboptions:*
	commitmenttype *(Keyword) Type of commitment associated with the signature within the scope of the specified policy. Supported keywords (default:* `none`*):*
	approval *The signer has approved the content of the message.*
	creation *The signer has created the message (but not necessarily approved, nor sent it).*
	delivery *The trusted service provider has delivered a message in a local store accessible to the recipient of the message.*
	none *No commitment type is included in the signature*
	origin *The signer recognizes to have created, approved, and sent the message.*
	receipt *The signer recognizes to have received the content of the message.*
	sender *The signer has sent the message (but not necessarily created it).*
	notice *(Text string) Human-readable text description of the signature policy*
	oid *(String; required) Object ID of the signature policy*
	uri *(String) URI of the signature policy*
prevent-changes	*(Boolean; only if* `certification` *is different from* `none`*) If* `true`*, the changes which are prohibited with the* `certification` *option (i.e. those changes which would invalidate the certification signature) are prevented in Acrobat, i.e. the respective tools are disabled in the user interface. Default:* `true`
reason	*(Text string; only relevant for* `digitalid`*; not allowed with* `policy`*) Reason for signing the document*
rootcertdir	*(String; not for* `engine=mscapi`*) Name of a directory which contains trusted root CA certificates in PEM encoding which may be required for validating the certificate chain. See »Naming convention for certificate and CRL files«, page 175, for file name conventions.*
rootcertfile	*(String; not for* `engine=mscapi`*) Name of a file which contains one or more trusted root CA certificates in PEM encoding which may be required for validating the certificate chain. For security reasons the file is not searched in the* `searchpath`*.*
rsaencoding	*(Keyword; not for* `engine=mscapi`*) Encoding method for RSA signatures (default: pkcs#1):*
	pkcs#1 *RSA encoding according to PKCS#1 v1.5. This method is supported in all Acrobat versions.*
	pss *RSA encoding according to PSS per RFC 3447/RFC 8017 which offers security advantages.*
signature	*(Boolean) If* `false` *no signature is created. This may be useful to switch between signing and other processing even if a prior call to prepare_signature( ) supplied signature options. Default:* `true`
sigtype	*(Keyword; only relevant for* `digitalid`*; not for* `engine=mscapi`*) Signature type (default:* `cades`*):*
	cms *CMS-based signature according to ISO 32000-1 and PAdES part 2 (ETSI TS 102 778-2)*
	cades *CAdES-based signature according to CAdES (ETSI TS 101 733) and RFC 5126. This is a requirement for PAdES part 3 and part 4.*

Table 8.7 Options for prepare_signature( )

option	description
timestamp	*(Option list or keyword; not for* `engine=mscapi`*) The signature includes an embedded timestamp created by a trusted timestamp authority (TSA). Supported suboptions (default:* `{source={ } critical=false}`*, i.e. the* `TimeStamp` *extension in the digital ID is used if present)*:
	critical *(Boolean; forced to* `true` *for document-level timestamps) If* `true`*, a signature is only generated if a valid timestamp can be obtained; otherwise an error is returned. If this option is* `false` *timestamping is silently ignored if no valid timestamp response can be obtained. Default:* `true`
	hash *(Keyword) Hash algorithm for creating the timestamp request. The algorithm must be supported by the TSA (default:* `sha256`*):* `sha1` *(not recommended),* `sha256`*,* `sha384`*, or* `sha512`
	policy *(String) OID of the TSA policy under which the timestamp must be created. Timestamping fails if the TSA does not support the specified policy.*
	source *(Network option list according to Table 8.11) Option list describing the TSA. The protocols* `http` *and* `https` *are supported.* *Only for embedded timestamps, but not for document-level timestamps: the* `url` *suboption of the* `source` *option or the the* `source` *option itself may be omitted which means that the* `TimeStamp` *extension in the digital ID is used.*
	The keyword `none` *means that no timestamp is embedded even if the* `TimeStamp` *extension is present in the signing certificate.*
timestamp-size	*(Integer) Estimated size of timestamp objects, used for reserving space for the CMS container of document timestamps and signature timestamps. Default: 7168*
update	*(Boolean) If* `true`*, signature data is appended as one or more incremental PDF update sections to a copy of the original document. Otherwise the PDF object hierarchy is rewritten which implies that existing signatures are lost. Validation information for embedded timestamps and document timestamps is always appended as update, regardless of this option. Update mode is not possible for input documents which require repair.* *Default:* `true` *but encryption forces the option value to* `false` *(i.e. the* `masterpassword` *or* `userpassword` *options of create_document( ) or a call to add_recipient( ) with a non-empty* `certificate` *option list)*
validate	*(Keyword) Control validation of involved certificates (default:* `full` *if* `ltv=full`*, otherwise* `formal`*):*
	formal *The following checks are applied:* *Critical extension flags, key usage etc. are checked;* *OCSP response is retrieved if requested and requires a valid response with status* `good`*;* *CRL is retrieved if requested and the signing certificate is checked against CRL; CRL date is checked;*
	full *Like* `validate=formal` *plus full validation of the certificate chain. This requires that all necessary root and intermediate CA certificates are available, as well as OCSP or CRL revocation information for all involved certificates (except for root certificates and an OCSP responders with the* `id-pkix-ocsp-nocheck` *extension).*

Table 8.8 Suboptions of the `digitalid` *option of prepare_signature( ) and open_document( ) as well as the* `certificate` *option of add_recipient( )*

option	*description*
Suboption for engine=builtin (if called from the `digitalid` ***or*** `certificate` ***option):***	
filename[1]	*(String; required) If called from the* `digitalid` *option: name of a disk-based or virtual digital ID file in PKCS#12 or PFX format (see Appendix A, »Working with Certificates« for conversion hints).* *If called from the* `certificate` *option: name of a disk-based or virtual X.509 certificate file in PEM or DER encoding. The certificate file must contain exactly one certificate.*
Suboptions for engine=pkcs#11 (only if called from the `digitalid` ***option):***	
signercert	*(String; requires exactly one of* `id` *or* `label`*) Name of the signer's certificate in DER encoding containing the public key for the corresponding private key on the PKCS#11 token. The private key must be selected via one of the options* `id` *or* `label`*. This option is required for HSMs which store only the private key, but not the corresponding certificate (e.g. AWS CloudHSM). Default: no certificate file name, i.e. the certificate must be available on the token along with the private key*
externalhash	*(Boolean) If* `true`*, the document hash for the signature is created via the PKCS#11 interface (i.e. on the token or HSM), otherwise it is created with the builtin engine. Default:* `false`
filename	*(String; required) Name of the PKCS#11 DLL/shared library for the cryptographic token. This must be a disk-based file, not a PVF file. Example:* `cryptoki.dll`
id	*(String; requires* `signercert`*) Select a private key for signing by its key identifier (PKCS#11 attribute CKA_ID). It must be provided as a decimal string or hexadecimal string (prefixed with 0x), e.g.* `id=0x03A247B2`.
initmode	*(Integer; only relevant for* `threadsafe=true`*) Initialization mode of the PKCS#11 library. The values correspond to the four modes described in the documentation of the PKCS#11 function* `C_Initialize( )`*, see 5.4 »General-purpose functions« in the PKCS#11 specification at docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html#_Toc416959540:* ***1*** *The application does not access the PKCS#11 library from multiple threads simultaneously although* `threadsafe=true` *was specified.* ***2*** *The application accesses the PKCS#11 library from multiple threads simultaneously, and the PKCS#11 library needs to use native operating system primitives to ensure safe multi-threaded access.* ***3*** *The application accesses the PKCS#11 library from multiple threads simultaneously, and the PKCS#11 library needs to use functions supplied by PLOP DS for mutex handling to ensure safe multi-threaded access.* ***4*** *The application accesses the PKCS#11 library from multiple threads simultaneously, and the PKCS#11 library can choose whether it uses functions supplied by PLOP DS or whether it uses native operating system primitives for mutex handling.* *The default mode 4 gives the PKCS#11 library the freedom to chose the locking method in multi-threaded environments. This option can be used as workaround if the PKCS#11 library requires another initialization mode for multi-threaded access.*
issuer	*(String) Select a digital ID by its* `issuer` *field (PKCS#11 attribute CKA_ISSUER). See option* `subject` *below for a description of the query format.*
label	*(String) Select a digital ID (or private key if* `signercert` *is specified) by its user-friendly label (PKCS#11 attribute CKA_ LABEL).*
serial	*(String) Select a digital ID by its serial number (PKCS#11 attribute CKA_SERIAL_NUMBER). The serial number must be provided as a decimal string or hexadecimal string (prefixed with* `0x`*), e.g.* `serial=0x03A247B2`
readonly-session	*(Boolean) If* `true`*, initiate a PKCS#11 read-only session for signing operations, otherwise a read/write session. Setting this option to* `false` *can be used as a workaround for non-conforming PKCS#11 libraries which require a read/write session. Default:* `true`

Table 8.8 Suboptions of the digitalid *option of prepare_signature( ) and open_document( ) as well as the* certificate *option of add_recipient( )*

option	description
slotid	*(Positive integer) Number of the slot that interfaces with the token. This can be used to directly select a slot if multiple slots are available.*
subject	*(String) Select a digital ID by its* subject *field (PKCS#11 attribute CKA_ SUBJECT). The query must be in the format* /type0=value0/type1=value1/...; *characters may be escaped by* \ *(backslash). The order of attributes is significant. If the token contains more than one digital ID the options* issuer, label, *and* subject *can be used for certificate selection.* *Example:* subject={/C=DE/L=Munich/O=PDFlib GmbH/CN=PLOP Demo Signer RSA-2048}
sticky	*(Boolean) If* true, *the PKCS#11 DLL/shared library remains loaded until the end of the process. This may offer performance advantages and may also be useful to work around problems in the DLL/shared library such as memory leaks in its initialization routine. However, no other PKCS#11 DLL/shared library can be loaded in the same process. Once this option has been set to* true, *subsequent calls within the same process must also supply* sticky=true *and the suboption* filename *will silently be ignored. If* false, *the PKCS#11 library is unloaded in the call to delete( ) for the last PLOP object which used this library. Default:* false
threadsafe	*(Boolean) If* true, *the PKCS#11 library must support thread-safe operation and is initialized in thread-safe mode. If the PKCS#11 library doesn't support thread-safe operation the call fails. If* false, *the PKCS#11 library is initialized in single-threaded mode which is only allowed for single-threaded applications. Default:* true
Suboptions for engine=mscapi (if called from the* digitalid ***or certificate ***option):***	
filename[1]	*(String; one of* filename *or* store *is required if called from prepare_signature( )*[2]*) Name of a disk-based or virtual digital ID file in PKCS#12 or PFX format (see Appendix A, »Working with Certificates« for conversion hints).*
storelocation	*(Keyword) Location of the certificate store (default:* current_user*):* current_service, current_user, current_user_group_policy, local_machine, local_machine_enterprise, local_machine_group_policy, services, users *The following locations can be opened remotely by prefixing the* store *option with the computer name (separated by a backslash character):* local_machine, local_machine_group_policy, services, users.
subject	*(String; required if called from prepare_signature( ) or add_recipient( ), and* store *is specified; ignored otherwise) Select a digital ID where the* subject *field contains the supplied string. It usually holds the* common name *(CN) field of the digital ID. This suboption is not required when called from open_document( ) since PLOP automatically determines the appropriate ID.*
store	*(String; one of* filename *or* store *is required if called from prepare_signature( )) Name of the certificate store, e.g.* My, Root, Trust. *If* storelocation=services *or* storelocation=users *the store name must be prefixed with the service or user name (separated by a backslash character). Default:* My

1. For security reasons the file is not searched in the searchpath *if called from the* certificate *option.*
2. If neither filename *nor* store *is supplied in prepare_signature( ) all available IDs in the default store are tried.*

Table 8.9 Suboptions of the `field` *option of prepare_signature( )*

option	*description*
fillexisting	*(Boolean; only relevant if one or more signature fields exist in the document and the* `name` *option is not supplied) If* `true`*, the first signature field in the input document is used for signing. If* `false`*, a new signature field is created with a unique name based on the pattern* `Signature#`*. This option is forced to* `true` *if the* `visdoc` *option is supplied in PDF/UA mode. Default:* `false`
name	*(Text string; must not end in a period ».« character) Name of an existing or new signature field. If the document contains a signature field with this name, it is used for the signature (and* `page` *is ignored), otherwise the field is created. If a field with this name exists, but has a type other than Signature, an error is thrown.* *Default: if no signature fields exist, a new one with the name* `Signature1` *is created. Otherwise field creation is controlled by the option* `fillexisting`*.*
page	*(Positive integer; ignored if an existing signature field is filled) Number of the page on which the signature field is created. The first page has number 1. Default: 1*
position	*(List with two Keywords) Relative position of the visualization page within the field. The visualization page is placed in the rectangle according to the supplied keywords and scaled such that it entirely fits into the rectangle while preserving its aspect ratio. The first keyword specifies the horizontal position with one of the values* `left`*,* `center`*,* `right`*; the second keyword specifies the vertical position with one of the values* `top`*,* `center`*,* `bottom`*. If both values are equal, it is sufficient to specify a single keyword. Default:* `{center}`
rect	*(Rectangle) Coordinates of the lower left and upper right corners of the signature field in PDF coordinates (one unit is 1/72 inch, origin at the lower left corner). The specified rectangle is completely filled with the visualization page. In order to avoid distortion the keyword* `adapt` *can be supplied instead of one or two coordinates. In this case the missing coordinate(s) are calculated automatically. At least one corner must be specified explicitly. The rectangle must not exceed the page. See »Location and size of the signature field«, page 103, for more details on the fitting process. An empty rectangle with four zero values implies an invisible field.* *Default: if an existing field is used its rectangle serves as default; otherwise an empty rectangle (i.e. invisible signature)*
tooltip	*(Non-empty text string) Text of the tooltip (also called alternative text) for a signature field. It may be used by screen readers to improve accessibility. Default: none*
visdoc	*(Document handle obtained with open_document( ); not allowed in PDF/X or PDF/VT mode; only allowed for a non-empty field rectangle and required in this case) Document from which a page is used for visualizing the signature on the page. In PDF/A mode the visualization document must be compatible to the generated output (see »PDF/A conformance«, page 104). In PDF/UA mode the input document must contain a suitable signature form field (see »PDF/UA conformance«, page 105).*
vispage	*(Integer; only relevant if* `visdoc` *is supplied) Page number in the document which is used for visualizing the signature (the first page has number 1). Default: 1*

Table 8.10 Suboptions of the ocsp *option of prepare_signature( )*

option	*description*
critical	*(Boolean; only relevant for* digitalid*) If* true*, a signature is only generated if a valid OCSP response for the signing certificate with status* good *was returned; otherwise an error is returned and no signature is created. If this option is* false *OCSP response embedding is silently ignored if no valid good OCSP response could be retrieved. Default:* true
freshness	*(Integer) Maximum amount of time in minutes after the OCSP response's* thisUpdate *entry for which a response is regarded as valid. If the response is older than the specified period (extended by* maxclockskew*) it is regarded as invalid and not used. Default: 1440 (1 day)*
hash	*(Keyword) Hash algorithm used to identify the certificate in all OCSP requests and responses. The algorithm must be supported by the OCSP responder (default:* sha1*):* sha1*,* sha256*,* sha384*, or* sha512 *Since Acrobat XI/DC support only* sha1 *all other values require* conformance=extended*.*
maxclockskew	*(Integer) Maximum tolerance in minutes when checking whether the response is fresh enough according to the* thisUpdate *entry in the OCSP response and the* freshness *option. This option can be used to compensate network delays or inaccurate system clocks. Default: 5*
nonce	*(Boolean) If* true*, the nonce extension (»number used only once«) is included in all OCSP requests, and the same value must be present in OCSP responses. Nonce handling prevents replay attacks, but also thwarts caching and is therefore not supported by some OCSP responders. Default:* true
source	*(Network option list) Option list describing a server from which an OCSP response for the signing certificate is requested and then embedded in the signature or DSS. The protocols* http *and* https *are supported. The* url *suboption of the* source *option or the* source *option itself can be omitted which means that the URL is taken from the* authorityInfoAccess *extension (AIA) in the digital ID.* *All suboptions of the* source *network option list except* url *and* httpauthentication *are also applied to OCSP requests for certificates other than the signing certificate. This facilitates the use of the same credentials (e.g.* username/password*) in OCSP calls for all involved certificates.*

Network option lists. Some digital signature features require access to a network resource such as TSAs and OCSP responders. The server and details for accessing it are specified in a network option list according to Table 8.11. The options in Table 8.7 and Table 8.10 which uses the data type »network option list« specify the list of supported protocols. Some examples for using network options lists (the network option list part is shown in blue):

```
timestamp={source={url={http://timestamp.acme.com/}} hash=sha384} digitalid=...

ocsp={source={url={http://ocsp.acme.com/}} } digitalid=...

ocsp={source={url={http://ocsp.acme.com/}
        proxy={url={http://user:password@proxy.company.com:8080}}} } digitalid=...

ocsp={source={timeout=1000}} digitalid=...
```

Table 8.11 Suboptions for a network option list

option	*description*
clientcert	*(Option list) Suboptions for configuring authentication with a client certificate. Only relevant when connecting to an HTTPS server. Supported suboptions:* ***certfile*** *(String) Name of the client certificate file in PEM format.* ***keyfile*** *(String) Name of the private key file in PEM format.* ***password*** *(String) Password for opening the private key file.*
httpauthentication	*(Keyword; only for* `http`*) Authentication method to try. A server may not support a particular authentication method (or any at all). Setting the authentication type explicitly may be preferable over the default (even if it results in the same method being selected) due to performance advantages. Supported keywords (default:* `any`*):* ***any*** *Select the most secure authentication method supported by the server.* ***anysafe*** *Like* `any`*, but exclude basic authentication.* ***basic*** *Basic authentication with user name and password. This method is not recommended since user name and password are sent over the network in plain text.* ***digest*** *Digest authentication with hashed user name and password according to RFC 2617.* ***ntlm*** *NTLM authentication as used in Microsoft products*
password	*(String) Password for basic and digest authentication*

Table 8.11 Suboptions for a network option list

option	*description*
proxy	*(Option list) Suboptions for configuring network access through a proxy server:* ***httpauthentication*** *(Keyword; only for* `http` *proxy) See main network option of the same name.* ***noproxy*** *(String) Comma-separated list of host names that do not require a proxy. A numerical IPv6 address must be specified without enclosing brackets.* ***password*** *(String) See main network option of the same name.* ***sslcertdir*** *(String; only for* `https` *proxy) See main network option of the same name.* ***sslcertfile*** *(String; only for* `https` *proxy) See main network option of the same name.* ***sslverifyhost*** *(String; only for* `https` *proxy) See main network option of the same name.* ***sslverifypeer*** *(String; only for* `https` *proxy) See main network option of the same name.* ***url*** *(String; required) Host name or numerical IP address of the proxy server. The URL may include user name and password. A numerical IPv6 address must be enclosed in brackets* `[...]`*. A port number may be appended with a colon ':' at the end. If no port number is specified the default port number 1080 is used. If no protocol is specified* `http` *is used.* ***username*** *(String) See main network option of the same name.* *A proxy server can also be configured with the common environment variables* `http_proxy`, `https_proxy`, `no_proxy`, `all_proxy`. *Options have priority over environment variables.*
sslcertdir	*(String; only for* `https`*) Name of a directory which contains trusted CA certificates in PEM encoding which may be required for establishing an SSL connection. See »Naming convention for certificate and CRL files«, page 175, for file name conventions.*
sslcertfile	*(String; only for* `https`*) Name of a file which contains one or more trusted CA certificates in PEM encoding which may be required for establishing an SSL connection.*
sslverifyhost	*(Boolean; only for* `https`*) If* `true`*, the Subject Alternate Name field in the server certificate must match the host name in the URL in order to establish an SSL connection. Default:* `true`
sslverifypeer	*(Boolean; only for* `https`*) If* `true`*, the server certificate must be verifiable against the set of trusted certificates supplied with the* `sslcertdir` *or* `sslcertfile` *options. If* `false`*, a server certificate which cannot be verified because no known trusted root is available for it is accepted. Default:* `true`
timeout	*(Integer) Timeout for accessing the resource in milliseconds. The value 0 means that no timeout is in effect. Default: 15000*
url	*(String; usually required, but optional for cases where the URL is known from context) Fully qualified URL of the network resource including the leading protocol identifier. The set of supported protocols is specified in the description of the respective option which deploys a network option list. Characters can be specified in URL encoding, e.g.* `%20`*. The URL may include user name and password, e.g.* `http://user:password@timestamp.acme.com/`
username	*(String) User name for basic and digest authentication*

8.7 Exception Handling

PLOP supplies auxiliary methods for handling library exceptions in the C language. Other PLOP language bindings use the native exception handling system of the respective language, such as *try/catch* clauses. The language wrappers pack information about exception number, description, and API method name into the generated exception object.

When a PLOP exception occurred, no other PLOP function except *delete( ), get_errnum( ), get_errmsg( ), get_apiname( )* may be called with the corresponding PLOP object.

The PLOP language bindings for Java and .NET define a separate *PLOPException* object which offers several members to access detailed error information.

C++ Java C# ***int get_errnum( )***
Perl PHP ***int get_errnum( )***
C ***int PLOP_get_errnum(PLOP *plop)***

Get the number of the last thrown exception, or the reason of a failed function call.

Returns The exception's error number.

Bindings In .NET this method is also available as *Errnum* in the *PLOPException* object.
In Java this method is also available as *get_errnum( )* in the *PLOPException* object.

C++ Java C# ***String get_errmsg( )***
Perl PHP ***string get_errmsg( )***
C ***const char *PLOP_get_errmsg(PLOP *plop)***

Get the descriptive text of the last thrown exception, or the reason of a failed function call.

Returns A string describing the error, or an empty string if the last API call didn't cause any error.

Bindings In .NET this method is also available as *Errmsg* in the *PLOPException* object.
In Java this method is also available as *getMessage( )* in the *PLOPException* object.

C++ Java C# ***String get_apiname( )***
Perl PHP ***string get_apiname( )***
C ***const char *PLOP_get_apiname(PLOP *plop)***

Get the name of the API method which threw the last exception or failed.

Returns The name of a PLOP API method.

Bindings In .NET this method is also available as *Apiname* in the *PLOPException* object.
In Java this method is also available as *get_apiname( )* in the *PLOPException* object.

C *PLOP_TRY(PLOP *plop)*

Set up an exception handling frame; must always be paired with *PLOP_CATCH( )*.

Details See »Error handling«, page 45.

C *PLOP_CATCH(PLOP *plop)*

Catch an exception; must always be paired with *PLOP_TRY( )*.

Details See »Error handling«, page 45.

C *PLOP_EXIT_TRY(PLOP *plop)*

Inform the exception machinery that a *PLOP_TRY( )* block will be left without entering the corresponding *PLOP_CATCH( )* clause.

Details See »Error handling«, page 45.

C *PLOP_RETHROW(PLOP *plop)*

Re-throw an exception to another handler.

Details See »Error handling«, page 45.

8.8 Global Options

C++ Java C# ***void set_option(String optlist)***

Perl PHP ***set_option(string optlist)***

C ***void PLOP_set_option(PLOP *plop, const char *optlist)***

Set one or more global options for PLOP.

optlist An option list specifying global options according to Table 8.12. If an option is provided more than once the last instance overrides all previous ones. In order to supply multiple values for a single option (e.g. *searchpath)* supply all values in a list argument to this option.

Details Multiple calls to this function can be used to accumulate values for those options marked in Table 8.12. For unmarked options the new value overrides the old one.

Table 8.12 Global options for set_option( )

option	***description***
filename-handling	*(Keyword) Indicates the encoding of file names. File names supplied as function parameters without UTF-8 BOM in non-Unicode aware language bindings are interpreted according to this option to guard against characters which would be illegal in the file system and to create a Unicode version of the file name. An error occurs if the file name contains characters outside the specified encoding. Default:* `unicode` *on Windows and macOS, otherwise* `honorlang`*:* ***ascii*** *7-bit ASCII* ***basicebcdic*** *Basic EBCDIC according to code page 1047, but only Unicode values <= U+007E* ***basicebcdic_37*** *Basic EBCDIC according to code page 0037, but only Unicode values <= U+007E* ***honorlang*** *The environment variables LC_ALL, LC_CTYPE and LANG are interpreted. The codeset specified in* `LANG` *is applied to file names if it is available.* ***unicode*** *Unicode encoding in (EBCDIC-) UTF-8 format* ***all names of 8-bit and CJK encodings*** *Any encoding recognized by PLOP*
license	*(String) Set the license key. It must be set before the first call to open_document*( ).*
licensefile	*(String) Set the name of a file containing the license key(s). The license file can be set only once before the first call to open_document*( ). Alternatively, the name of the license file can be supplied in an environment variable called PDFLIBLICENSEFILE or (on Windows) via the registry.*
frontpage	*(Boolean) If* `false`*, an exception is thrown if no valid license key was found; if* `true`*, a front page is created in evaluation mode according to Section 0.1, »Installing the Software«, page 7. This option must be set before the first call to open_document*( ). It doesn't have any effect if a valid license key was found. Default:* `true`
logging[1]	*(Option list; unsupported) An option list specifying logging output according to Table 8.14. Alternatively, logging options can be supplied in an environment variable called PLOPLOGGING or on Windows via the registry. An empty option list will enable logging with the options set in previous calls. If the environment variable is set logging will start immediately after the first call to new( ).*
mmiolimit	*(Integer) Upper limit for the size of input files in MB (=1024*1024 bytes) which will be memory-mapped. Setting this option to 0 (zero) disables memory mapping. Disabling memory mapping can be used on non-Windows systems to avoid problems when remote files suddenly become unavailable while being used. Default: 50 on 32-bit platforms, 2048 otherwise*

Table 8.12 Global options for set_option( )

option	description
searchpath[1]	*(List of name strings) Relative or absolute path name(s) of a directory containing files to be read. The search path can be set multiply; the entries will be accumulated and used in least-recently-set order. It is recommended to use double braces even for a single entry to avoid problems with directory names containing space characters. An empty string list (i.e.* {{}} *) deletes all existing search path entries including the default entries. On Windows the searchpath can also be set via a registry entry. Default: empty*
userlog	*(Name string) Arbitrary string which will be written to the log file if logging is enabled.*

1. Option values can be accumulated with multiple calls.

8.9 Logging

The logging feature can be used to trace API calls. The contents of the log file may be useful for debugging purposes or may be requested by PDFlib GmbH support. Table 8.13 lists the options for activating the logging feature with *set_option( )*.

Table 8.13 Logging-related keys for set_option( )

key	explanation
logging	*Option list with logging options according to Table 8.14*
userlog	*String which will be copied to the log file*

The logging options can be supplied in the following ways:

- As an option list for the *logging* option of *set_option( )*, e.g.:

```
plop.set_option("logging={filename={debug.log} remove}");
```

- In an environment variable called *PLOPLOGGING*. Doing so will activate the logging output starting with the very first call to one of the API methods.
- (Unsupported) In the following registry key:

```
HKLM\SOFTWARE\PDFlib\PLOP5\PLOPLOGGING
```

Table 8.14 Suboptions for the `logging` *option of set_option( )*

option	explanation	
classes	*(Option list) Option list where each option describes a logging class, and the corresponding value describes the level. Level 0 disables a logging class, positive numbers enable a class. Increasing levels provide more detailed output. If no level is mentioned for a class the value 1 must be used (initial value: api=1).*	
	api	*Log all API calls with their function parameters and results. If* `api=2` *a timestamp will be created in front of all API trace lines, and deprecated functions and options will be marked.*
	convert	*String conversion.*
	digsig	*Log details about digital signature creation:* **1** *basic information* **2** *validation information; OCSP and CRL details; PKCS#11 library, slot, and token info* **5** *certificate details*
	filesearch	*Log all attempts related to locating files via* `SearchPath` *or PVF.*
	network	*Log details about network activity:* **1** *general network information* **2** *network headers and statistics* **3** *detailed network data*
	resource	*Log all attempts at locating resources via Windows registry, UPR definitions as well as the results of the resource search.*
	user	*User-specified logging output supplied with the* `userlog` *option.*
	warning	*Log all warnings, i.e. error conditions which can be ignored or fixed internally. If* `warning=2` *messages from functions which do not throw any exception, but hook up the message text for retrieval via get_errmsg( ), and the reason for all failed attempts at opening a file (searching for a file in searchpath) will also be logged.*
disable	*(Boolean) Disable logging output. Default:* `false`	

Table 8.14 Suboptions for the `logging` *option of set_option( )*

option	*explanation*
filename	*(String) Name of the log file (stdout and stderr are also acceptable). Output will be appended to any existing contents. The log file name can alternatively be supplied in an environment variable called PLOPLOGFILENAME (in this case the option* `filename` *will be ignored). Default:* `plop.log` *(on Windows and macOS in the / directory, on Unix in /tmp)*
flush	*(Boolean) If* `true`*, the log file will be closed after each output, and reopened for the next output to make sure that the output will actually be flushed. This may be useful when chasing program crashes where the log file is truncated, but significantly slows down processing. If false, the log file will be opened only once. Default:* `false`
includepid	*(Boolean; not on MVS) Include the process id in the log file name. This should be enabled if multiple processes use the same log file name. Default:* `false`
includetid	*(Boolean; not on MVS) Include the thread id in the log file name. This should be enabled if multiple threads in the same process use the same log file name. Default:* `false`
includeoid	*(Boolean; not on MVS) Include the object id in the log file name. This should be enabled if multiple PLOP objects in the same thread use the same log file name. Default:* `false`
remove	*(Boolean) If* `true`*, an existing log file will be deleted before writing new output. Default:* `false`
removeonsuccess	*(Boolean) Remove the generated log file in delete( ) unless an exception occurred. This may be useful for analyzing occasional problems in multi-threaded applications or problems which occur only sporadically. It is recommended to combine this option with* `includepid`/`includetid`/`includeoid` *as appropriate.*
restore	*(Boolean) Restore the state of all logging class levels (except those specified in the same option list) to the least recently saved state.*
save	*(Boolean) Save the state of all logging class levels (except those specified in the same option list). Up to 7 save levels are supported.*
stringlimit	*(Integer) Limit for the number of characters in text strings, or 0 for unlimited. Default: 0*

8.10 pCOS Functions

The full pCOS syntax for retrieving object data from a PDF is supported; see the pCOS Path Reference for a detailed description.

C++ Java C# ***double pcos_get_number(int doc, String path)***

Perl PHP ***double pcos_get_number(long doc, string path)***

C ***double PLOP_pcos_get_number(PLOP *plop, int doc, const char *path, ...)***

Get the value of a pCOS path with type *number* or *boolean*.

doc A valid document handle obtained with *open_document*( )*.

path A full pCOS path for a numerical or boolean object.

Additional parameters (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders *(%s* for strings or *%d* for integers; use *%%* for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

Returns The numerical value of the object identified by the pCOS path. For Boolean values 1 will be returned if they are *true,* and 0 otherwise.

C++ Java C# ***string pcos_get_string(int doc, String path)***

Perl PHP ***string pcos_get_string(long doc, string path)***

C ***const char *PLOP_pcos_get_string(PLOP *plop, int doc, const char *path, ...)***

Get the value of a pCOS path with type *name, number, string,* or *boolean*.

doc A valid document handle obtained with *open_document*( )*.

path A full pCOS path for a string, number, name, or boolean object.

Additional parameters (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders *(%s* for strings or *%d* for integers; use *%%* for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

Returns A string with the value of the object identified by the pCOS path. For Boolean values the strings *true* or *false* will be returned.

Details This function raises an exception if pCOS does not run in full mode and the type of the object is *string*. However, the objects */Info/** (document info keys) can also be retrieved in restricted pCOS mode if *nocopy=false* or *plainmetadata=true*, and *bookmarks[...]/Title* as well as all paths starting with *pages[...]/annots[...]/* can be retrieved in restricted pCOS mode if *nocopy=false*.

This function assumes that strings retrieved from the PDF document are text strings. String objects which contain binary data should be retrieved with *pcos_get_ stream( )* instead which does not modify the data in any way.

Bindings C language binding: The string is returned in UTF-8 format without BOM (on IBM Z: EBCDIC-UTF8). The returned strings are stored in a ring buffer with up to 10 entries. If more than 10 strings are queried, buffers will be reused, which means that clients must copy the strings if they want to access more than 10 strings in parallel. For example, up to 10 calls to this function can be used as parameters for a *printf( )* statement since the return strings are guaranteed to be independent if no more than 10 strings are used at the same time.

Java and .NET: the result is provided as Unicode string.

Perl, PHP, Python and Ruby language bindings: the result is provided as UTF-8 string.

C++ ***const unsigned char *pcos_get_stream(int doc, int *len, string optlist, wstring path)***

C# Java ***final byte[ ] pcos_get_stream(int doc, String optlist, String path)***

Perl PHP ***string pcos_get_stream(long doc, string optlist, string path)***

C ***const unsigned char *PLOP_pcos_get_stream(PLOP *plop, int doc, int *len, const char *optlist, const char *path, ...)***

Get the contents of a pCOS path with type *stream*, *fstream*, or *string*.

doc A valid document handle obtained with *open_document*( )*.

len (C and C++ language bindings only) A pointer to a variable which will receive the length of the returned stream data in bytes.

optlist An option list specifying stream retrieval options according to Table 8.15.

path A full pCOS path for a stream or string object.

Additional parameters (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders *(%s* for strings or *%d* for integers; use *%%* for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

Returns The unencrypted data contained in the stream or string. The returned data will be empty (in C and C++: NULL) if the stream or string is empty, or if the contents of encrypted attachments in an unencrypted document are queried and the attachment password has not been supplied.

If the object has type *stream,* all filters will be removed from the stream contents (i.e. the actual raw data will be returned). If the object has type *fstream* or *string* the data will be delivered exactly as found in the PDF file, with the exception of ASCII85 and ASCIIHex filters which will be removed.

Details This function will throw an exception if pCOS does not run in full mode. As an exception, the object */Root/Metadata* can also be retrieved in restricted pCOS mode if *nocopy=false* or *plainmetadata=true*. An exception will also be thrown if *path* does not point to an object of type *stream*, *fstream*, or *string*.

Despite its name this function can also be used to retrieve objects of type *string*. Unlike *pcos_get_string( )*, which treats the object as a text string, this function will not modify the returned data in any way. Binary string data is rarely used in PDF, and cannot be reliably detected automatically. The user is therefore responsible for selecting the appropriate function for retrieving string objects as binary data or text.

Bindings C language binding: If *convert=unicode* is supplied, the string is returned in UTF-8 format without BOM (on IBM Z: EBCDIC-UTF8).

C and C++ language bindings: The returned data buffer can be used until the next call to this function.

Table 8.15 Options for pcos_get_stream( )

option	description	
convert	*(Keyword; will be ignored for streams which are compressed with unsupported filters) Controls whether or not the string or stream contents will be converted (default:* none*)* **:**	
	none	*Treat the contents as binary data without any conversion.*
	unicode	*Treat the contents as textual data (i.e. exactly as in pcos_get_string( )), and normalize it to Unicode. In non-Unicode-capable language bindings this means the data is converted to UTF-8 format without BOM.* *This option is required for the data type »text stream« in PDF which is rarely used (e.g. it can be used for JavaScript, although the majority of JavaScripts is contained in string objects, not stream objects).*

8.11 Unicode Conversion Function

C++ ***string convert_to_unicode(wstring inputformat, string input, wstring optlist)***

C# Java ***string convert_to_unicode(String inputformat, byte[ ] input, String optlist)***

Perl PHP ***string convert_to_unicode(string inputformat, string input, string optlist)***

C ***const char *PLOP_convert_to_unicode(PLOP *p,***
const char *inputformat, const char *input, int inputlen, int *outputlen, const char *optlist)

Convert a string in an arbitrary encoding to a Unicode string in various formats.

inputformat Unicode text format or encoding name specifying interpretation of the input string:

- Unicode text formats: *utf8, ebcdicutf8 (on EBCDIC platforms), utf16, utf16le, utf16be, utf32*
- All internally known 8-bit encodings, encodings available on the host system, and the CJK encodings *cp932, cp936, cp949, cp950*
- The keyword *auto* specifies the following behavior: if the input string contains a UTF-8 or UTF-16 BOM it will be used to determine the appropriate format, otherwise the current system codepage will be assumed.

input String to be converted to Unicode.

inputlen (C language binding only) Length of the input string in bytes. If *inputlen = 0* a null-terminated string must be provided.

outputlen (C language binding only) C-style pointer to a memory location where the length of the returned string (in bytes) will be stored.

optlist An option list specifying options for input interpretation and Unicode conversion:

- Input filter options according to Table 8.16: *charref, escapesequence*
- Unicode conversion options according to Table 8.16: *bom, errorpolicy, inflate, outputformat*

Returns A Unicode string created from the input string according to the specified parameters and options. If the input string does not conform to the specified input format (e.g. invalid UTF-8 string) an empty output string will be returned if *errorpolicy=return*, and an exception will be thrown if *errorpolicy=exception*.

Details This function may be useful for general Unicode string conversion. It is provided for the benefit of users working in environments which do not provide suitable Unicode converters.

Bindings C binding: the returned strings will be stored in a ring buffer with up to 10 entries. If more than 10 strings are converted, the buffers will be reused, which means that clients must copy the strings if they want to access more than 10 strings in parallel. For example, up to 10 calls to this function can be used as parameters for a *printf( )* statement since the return strings are guaranteed to be independent if no more than 10 strings are used at the same time.

Table 8.16 Options for convert_to_unicode( )

option	description
bom	*(Keyword; ignored for* `outputformat=utf32`*; in .NET, Java, Objective-C and Python only* `none` *is allowed*) *Policy for adding a byte order mark (BOM) to the output string. Supported keywords (default:* `none`*):* ***add*** *Add a BOM.* ***keep*** *Add a BOM if the input string has a BOM.* ***none*** *Don't add a BOM.* ***optimize*** *Add a BOM except if* `outputformat=utf8` *or* `ebcdicutf8` *and the output string contains only characters in the range < U+007F.*
charref	*(Boolean) If* `true`*, enable substitution of numeric and character entity references and glyph name references. Default:* `false`
errorpolicy	*(Keyword) Behavior in case of conversion errors (default:* `exception`*):* ***return*** *The replacement character will be used if a character reference cannot be resolved. An empty string will be returned in case of conversion errors.* ***exception*** *An exception will be thrown in case of conversion errors.*
escape-sequence	*(Boolean) If* `true`*, enable substitution of escape sequences in strings. Default:* `false`
inflate	*(Boolean; only for* `inputformat=utf8`*; will be ignored if* `outputformat=utf8`*) If* `true`*, an invalid UTF-8 input string will not trigger an exception, but rather an inflated byte string in the specified output format will be generated. This may be useful for debugging. Default:* `false`
output-format	*(Keyword) Unicode text format of the generated string:* `utf8`*,* `ebcdicutf8` *(on EBCDIC platforms),* `utf16`*,* `utf16le`*,* `utf16be`*,* `utf32`*. An empty string is equivalent to* `utf16`*. Default:* `utf16` *Unicode-capable language bindings: the output format is forced to* `utf16`*.* *C++ language binding: only the following output formats are allowed:* `utf8`*,* `utf16`*,* `utf32`**.**

A Working with Certificates

In this appendix we provide additional information which may be useful when working with certificates and PLOP or PLOP DS.

Display contents of a certificate. You can display the contents of a certificate or digital ID in PKCS#12 format with the following Windows command:

```
certutil -dump -p demo demo_signer_rsa_2048.p12
```

Display the contents of a certificate in PEM encoding with OpenSSL:

```
openssl x509 -inform PEM -in demo_recipient_1.pem -noout -text
```

Extract the public key from a digital ID to create a certificate with OpenSSL. If you have a digital ID in PKCS#12 format (with public and private key) and need a corresponding certificate in PEM encoding (with the public key only) for others to encrypt documents, you can use the following command which will prompt for the password:

```
openssl pkcs12 -in demo_recipient_1.p12 -clcerts -nokeys -out ↵
        demo_signer_rsa_2048.pem
```

Convert a certificate to PEM with OpenSSL. The signature options *certfile* and *rootcertfile* as well as the suboption *sslcertfile* in a network option list accept certificates only in the text-based PEM encoding. On EBCDIC platforms PEM certificates must be encoded in EBCDIC.

You can use the following OpenSSL command to convert certificates in the binary DER encoding to the required text-based PEM encoding:

```
openssl x509 -inform DER -outform PEM -in PDFlibDemoCA_G3.crt -out PDFlibDemoCA_G3.pem
```

Note that *.cer* and *.crt* files may use DER or PEM encoding. Since the file name suffix is not reliable, you can check the format in a text editor: while DER encoding consists of binary data, PEM encoding is a text-based format with Base-64-encoded data enclosed by the lines

```
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----
```

Naming convention for certificate and CRL files. The signature options *crldir* and *rootcertdir* as well as the suboption *sslcertdir* in a network option list accept the name of a directory where certificates or CRLs are searched. These files must be stored in the text-based PEM encoding and must be named according to the OpenSSL 1.0.0 (or above) file name hashing convention.

The OpenSSL command *c_rehash* creates symbolic links with the required hashed file names for one or more directories containing certificates or CRLs in PEM encoding:

```
c_rehash .
```

If you need to create hashed file names manually apply the following steps:

- Create the file name hashes for individual certificate or CRL files with an OpenSSL command similar to one the following:

```
# create the hashed file name for a certificate in PEM encoding
openssl x509 -hash -noout -in PDFlibDemoCA_G3.pem

# create the hashed file name for a certificate in DER encoding
openssl x509 -hash -noout -inform DER -in PDFlibDemoCA_G3.crt

# create the hashed file name for a CRL in PEM encoding
openssl crl -hash -noout -in PDFlibDemoCA_G3.crl.pem

# create the hashed file name for a CRL in DER encoding
openssl crl -hash -noout -inform DER -in PDFlibDemoCA_G3.crl
```

- Append a '.' (period) character. For CRLs also append the character 'r';
- Append the decimal number 0 (zero). If there is a conflict within the hashed file names in a directory increase the number by one.

Note OpenSSL on IBM Z cannot be used for calculating compatible file name hashes due to a bug.

Syntax for object identifiers (OIDs). The *oid* suboption of the *policy* option and the *policy* suboption of the *timestamp option* expect an object identifier (OID) which specifies the policy. OIDs consist of a series of decimal numbers where the numbers are separated by whitespace or period characters ».«, e.g.

```
2.16.840.1.101.3.2.1.48.9
```

B Combining PDFlib with PLOP DS

PLOP DS has been designed for easy interoperability with PDFlib for dynamically generating and signing PDF documents. In this appendix we discuss how you can combine both products.

File-Based Combination. The file-based method is recommended if you deal with very large PDF documents, or if you need to reduce the total memory requirements of the PDFlib/PLOP DS combination. Simply generate a PDF file on disk with appropriate PDFlib routines, and process the generated document with *open_document( )*.

Create documents in memory and digitally sign. The memory-based method is faster, but requires more memory. The following process is recommended for dynamic PDF generation and signature in Web applications unless you deal with very large documents:

- Instead of generating a PDF file on disk with PDFlib, use in-core PDF generation by supplying an empty file name to *PDF_begin_document( )*.
- Fetch the generated PDF data by calling *PDF_get_buffer( )* after *PDF_end_document( )*.
- Create a virtual file in PLOP based on the PDF data in memory by calling *create_pvf( )*.
- Pass the name of the PVF file to PLOP DS using *open_document( )*.

The *hellosign* programming sample which is included in all PLOP packages demonstrates how to use PDFlib for dynamically creating a PDF document and passing it to PLOP in memory for applying a digital signature.

Dynamically create signature visualization documents. You can also use PDFlib to dynamically create a document which is used for signature visualization (see Section 7.3.1, »Visualizing Signatures with a Graphic or Logo«, page 102). This is useful if you need to include varying text or image components in the visualization document, e.g. the current date/time.

The *dynamicsign* programming sample demonstrates how to use PDFlib for dynamically creating a PDF visualization document and pass it to PLOP DS for use in the signature creation process.

Form fields created with PDFlib 9.2 and earlier. When Acrobat opens a document containing form fields it automatically stores a visual representation of the fields in the PDF document if required. PDFlib 9.2 and earlier relies on this behavior and doesn't create the so-called appearance streams. However, since automatic appearance creation in Acrobat modifies the document immediately upon opening it, such documents are unsuitable for digital signing.

PLOP DS therefore rejects such documents by default (when signing in update mode). In rewrite mode, i.e. with *update=false*, such documents can be signed by supplying the option *sacrifice={fields}*. However, form fields in the input document are no longer present in the signed output when using this option.

There are no restrictions for form field documents created with PDFlib 9.3 or above.

Annotations created with PDFlib. If you use PDFlib to create documents with annotations and subsequently sign the documents with PLOP DS, Acrobat displays the signature as valid. However, Acrobat may display *»Annotations Modified«* (sometimes only af-

ter validating the signatures). The reason is that upon opening the document Acrobat silently adds a name entry to annotations that don't already have one. Acrobat also adds an »appearance stream«, i.e. a visual representation of the appearance. These modifications trigger the warning message regarding modifications in the document. In order to avoid this confusing message do the following when creating the input document with PDFlib:

- Supply the annotation name when creating the annotation. This can be achieved with the *name* option of the PDFlib API method *PDF_create_annotation( )*.
- Supply an appearance stream (i.e. a template) with the *template* option and suboption *normal* of the PDFlib API method *PDF_create_annotation( )*.

C PLOP Library Quick Reference

The following tables contain an overview of all PLOP API methods.

General Methods

Function prototype	*page*
*(C only) PLOP *PLOP_new(void)*	*138*
void delete( )	*138*
void create_pvf(String filename, byte[ ] data, String optlist)	*138*
int delete_pvf(String filename)	*139*
double info_pvf(String filename, String keyword)	*140*

Document Input and Output

Function prototype	*page*
int open_document(String filename, String optlist)	*141*
*(C only) int PLOP_open_document_callback(PLOP *plop, void *opaque, plop_off_t filesize, size_t (*readproc)(void *opaque, void *buffer, size_t size), int (*seekproc)(void *opaque, plop_off_t offset), const char *optlist)*	*143*
int create_document(String filename, String optlist)	*145*
void close_document(int doc, String optlist)	*144*
byte[ ] get_buffer( )	*148*
int prepare_signature(String optlist)	*152*

Error Handling

Function prototype	*page*
int get_errnum( )	*163*
String get_errmsg( )	*163*
String get_apiname( )	*163*

Global Options

Function prototype	*page*
void set_option(String optlist)	*165*

pCOS Methods

Function prototype	*page*
double pcos_get_number(int doc, String path)	*169*
string pcos_get_string(int doc, String path)	*169*
final byte[ ] pcos_get_stream(int doc, String optlist, String path)	*170*

Unicode Conversion

Function prototype	*page*
string convert_to_unicode(String inputformat, byte[ ] input, String optlist)	*172*

D Revision History

Revision history of this manual

Date	*Changes*
February 27, 2023	► *Updates for PLOP 5.5 and PLOP DS 5.5*
May 04, 2020	► *Updates for PLOP 5.4 and PLOP DS 5.4*
August 28, 2018	► *Changes for PLOP 5.3 and PLOP DS 5.3*
December 11, 2017	► *Minor updates for PLOP DS 5.2p2: clarify EC encryption support in Acrobat; sign PDFlib-created documents containing annotations*
February 17, 2017	► *Minor updates for PLOP 5.2 and PLOP DS 5.2*
September 30, 2016	► *Changes for certificate security and minor new functionality in PLOP and PLOP DS 5.1r1*
May 13, 2016	► *Changes for PLOP 5.1 and PLOP DS 5.1*
March 27, 2015	► *Minor corrections for PLOP and PLOP DS 5.0 r1*
December 04, 2014	► *Changes for PLOP 5.0 and PLOP DS 5.0*
September 09, 2014	► *Changes for PLOP 5.0 and PLOP DS 5.0 Beta 1*
March 13, 2014	► *Changes for PLOP 5.0 and PLOP DS 5.0 Alpha 2*
January 30, 2014	► *Changes for PLOP 5.0 and PLOP DS 5.0 Alpha 1*
March 04, 2011	► *Major overhaul for PLOP 4.1 and PLOP DS 4.1*
December 05, 2008	► *Updates for XMP, PVF, and PKCS#11 (smartcard) support in PLOP 4.0 and PLOP DS 4.0*
July 15, 2007	► *Updates for PLOP 3.0 and PLOP DS 3.0*
September 27, 2004	► *Updates for PLOP 2.1*
December 01, 2003	► *Updated for new major release PLOP 2.0*
November 23, 2002	► *Added a description of the Perl binding for PSP*
November 7, 2002	► *Added a section on the use of PSP with ILE-RPG*
October 22, 2002	► *Minor changes for PSP 1.0.1*
September 17, 2002	► *First edition for PSP 1.0.0*

Index

R

S

T

U

V

W

X

PDFlib GmbH

PDFlib GmbH
Franziska-Bilek-Weg 9
80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0

Licensing contact
sales@pdflib.com

Support
PDFlib_support@apryse.com *(please include your license number)*