

# Generic CMP client library API

– released, updated –

Version 1.**5  
corresponding to genCMPClient v0.93**

Dr. David von Oheimb, Siemens T RDA CST SEA-DE

Restricted

Copyright **** Siemens AG, 2018-2021

Table of Contents

1. Introduction 4

1.1 Motivation and Purpose 4

1.2 Scope 4

1.3 References 5

2. Overview 5

3. Core functionality 7

3.1 CMPclient\_prepare 7

3.2 CMPclient\_setup\_HTTP 9

3.3 CMPclient\_imprint, CMPclient\_bootstrap, and CMPclient\_pkcs10 9

3.4 CMPclient\_update 10

3.5 CMPclient\_revoke 11

3.6 CMPclient\_finish 11

3.7 Logging callback function 11

Message transfer callback function 12

3.8 Certificate checking callback function 12

3.9 Component credentials 13

4. Support functionality 14

4.1 CREDENTIALS helpers 14

4.2 X509\_STORE helpers 15

4.3 EVP\_PKEY helpers 17

4.4 SSL\_CTX helpers 17

4.5 X509\_EXTENSIONS helpers 18

5. Appendix: C header file 19

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Docum. Version** | **Author** | **Date** | **Changes** | **Reason for Change** |
| V0.1 | D. von Oheimb | 2018-02-27 | Initial structure & contents |  |
| V0.2 | D. von Oheimb | 2018-02-28 | Completed core decls | Review by H. Brockhaus |
| V0.6 | D. von Oheimb | 2018-03-19 | Sections 2 - 5 | Comments by DF&PD and during meeting |
| V0.7 | D. von Oheimb | 2018-03-20 | Sections 2 - 5 | Comments by BT and MO |
| V0.95 | D. von Oheimb | 2018-04-09 | Sections 1.2 - 4.3 | Feedback by DF&PD |
| V1.0 | D. von Oheimb | 2018-04-12 | None | Approval |
| V1.1 | D. von Oheimb | 2018-07-12 | Various improvements of contents and presentation | Experience implementing this API and an example |
| V1.2 | D. von Oheimb | 2018-11-07 | Various updates and minor improvements | changes in SecUtils and CMPforOpenSSL |
| V1.3 | D. von Oheimb | 2019-04-11 | All sections | Reflect latest API and code enhancements, minor description improvements also according to review |
| V1.4 | D. von Oheimb | 2020-04-30 | All sections | Reflect enhancements mostly on certificate status checking, credentials helpers, and HTTP(S) transport |
| V1.5 | D. von Oheimb | 2020-08-25   2021-07-14 | Sections 3.1 and 4.4   Sections 1, 3.2, 3.7, and 5 | Minor corrections and extensions w.r.t. cert chain building Update info on OpenSSL and logging; add keep\_alive for HTTP |

# Introduction

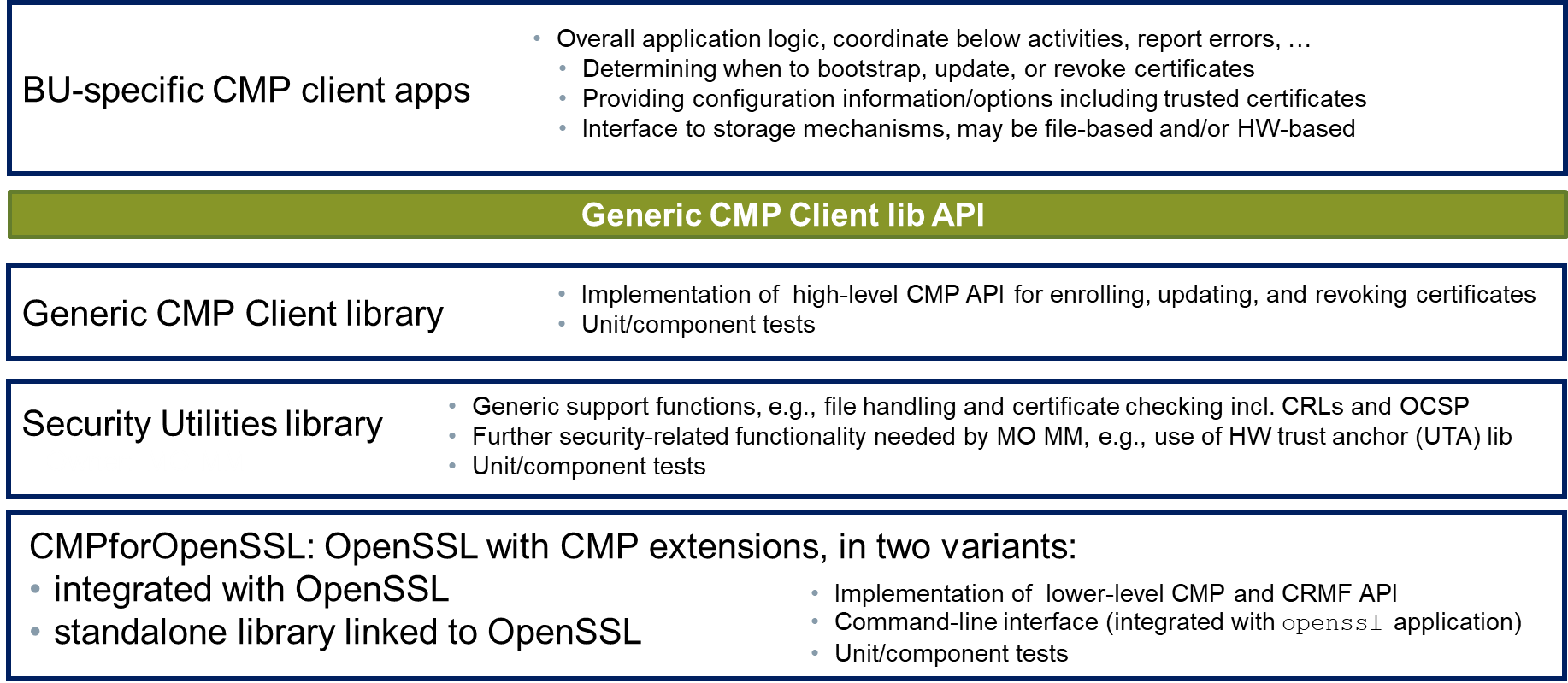
## Motivation and Purpose

CMP [RFC4210], including CRMF [RFC4211] and HTTP transfer for CMP [RFC6712], is used as certificate management protocol within Siemens for certificate management use cases such as enrollment, update and revocation of certificates. An overview and general information on CMP can be found on the Product PKI Wiki [CMP\_Wiki].

This document describes the Application Programming Interface (API) for the development of a generic CMP client library to be used by Siemens Companies and Business Units (BUs) in the development of their products and solutions.

## Scope and Status

There is an open-source CMP client implementation that we build upon: CMPforOpenSSL, located on GitHub [cmpossl]. It offers both a library based on OpenSSL with a low-level C API and a high-level command-line interface (CLI).



Goal when designing this interface was to offer a medium-level API based on the low-level CMPforOpenSSL API that on the one hand is convenient to use for application programmers and on the other hand is complete and flexible enough to cover the major certificate management use cases of the Siemens BUs. Besides its generic character, the library supports developing specific CMP clients that fulfill the CMP Profile for Industrial Certificate Management Use Cases [CMP\_Profile].

The implementation makes use of the SecurityUtilities library developed at Siemens Mobility and of CMPforOpenSSL, which can be provided either as an integrated extension to OpenSSL 3.0 or as a standalone CMP library dynamically linked to OpenSSL 1.0.2., 1.1.0, and 1.1.1. For details, history, and more current information please refer to the Wiki page of [cmpossl].

The library has been OSS-cleared and is going to be open-sourced by September 2021.

On top of the API a rather extensive CMP client CLI is available as demo application and for testing and exploration purposes.

Instructions for obtaining, building, installing, and using the library and the CLI can be found at [genCMPClient].

## References

| Reference ID | Document with title, unique identifier and version |
| --- | --- |
| [RFC4210] | RFC4210: “Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)“, 2005  <https://tools.ietf.org/html/rfc4210> |
| [RFC4211] | RFC4211: “Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)”, 2005  [https://www.ietf.org/RFC/RFC4211.txt](https://www.ietf.org/rfc/rfc4211.txt) |
| [RFC5280] | RFC5280: “Certificate and Certificate Revocation List (CRL) Profile”, 2008  [https://www.ietf.org/RFC/RFC5280.txt](https://www.ietf.org/rfc/rfc5280.txt) |
| [RFC6712] | RFC6712: “Internet X.509 Public Key Infrastructure -- HTTP Transfer for the Certificate Management Protocol (CMP)”, 2012  [https://tools.ietf.org/RFC/RFC6712.txt](https://tools.ietf.org/rfc/rfc6712.txt) |
| [OID-ref] | OID database, reference record for OID 1.3.6.1.5.5.7.3, 2019 <http://oidref.com/1.3.6.1.5.5.7.3> |
| [OpenSSL-ciphers] | OpenSSL manual page on SSL/TLS ciphers, 2019 <https://www.openssl.org/docs/manmaster/man1/ciphers.html> |
| [OpenSSL-config] | OpenSSL manual page on SSL/TLS ciphers, 2019 <https://www.openssl.org/docs/manmaster/man5/x509v3_config.html> |
| [OpenSSL-engine] | OpenSSL manual page on the crypto engine API, 2019 <https://www.openssl.org/docs/manmaster/man3/ENGINE_init.html> |
| [OpenSSL-sec-level] | OpenSSL manual page on SSL/TLS security level, 2019 <https://www.openssl.org/docs/manmaster/man3/SSL_CTX_get_security_level.html> |
| [cmpossl] | M. Peylo and D. von Oheimb: CMP for OpenSSL project, 2019  <https://github.com/mpeylo/cmpossl> (code repository including documentation) |
| [genCMPClient] | D. von Oheimb: Generic CMP Client library and CLI <https://code.siemens.com/product-pki/genCMPClient> |
| [CMP\_Wiki] | General information on CMP on the CT PKI Product PKI Wiki, 2019  <https://wiki.ct.siemens.de/x/zYPdBw> |
| [CMP\_Profile] | CMP Profile for Industrial Certificate Management Use Cases, V2.0, 2018 and IETF drafts, 2020 <https://wiki.ct.siemens.de/x/UCfsBw> |
| [Cert\_Valid] | Basic Certificate Validation Guideline for Certificate Management, V1.0, 2017 <https://wiki.ct.siemens.de/x/UCfsBw> |
| [TLS\_Conf] | “Transport Layer Security – Configuration Best Practice Guideline”, 2016  <https://wiki.ct.siemens.de/x/fJJfBw> |
| [TLS\_Integration] | “Transport Layer Security – Integration Best Practice Guideline”, 2016  <https://wiki.ct.siemens.de/x/eCZ-Bw> |

# Overview

This CMP client library is built on top of CMPforOpenSSL, which in turn is built on top of OpenSSL 1.x and has been integrated with OpenSSL 3.0. Thus we inherit several declarations from there. In particular, the type **CMP\_CTX** defining the context data structure for the low-level CMPforOpenSSL client library functions that will be re-used in the medium-level API.

For best usability and flexibility, we condense the CMP client core functionality into a few rather high-level functions that allow setting all typically required use case parameters. Each CMP use case can be executed by calling several of these functions to form a CMP client transaction as described next – see also the example code at the end of this section.

In order to enable the use of the low-level CMPforOpenSSL functions we directly re-use their CMP context data structure (of type **CMP\_CTX**). This context includes all parameters and state information of a CMP transaction. The function, **CMPclient\_prepare()** sets up those parts of the context data structure generally needed for all use cases. It must be called first when starting a new transaction. As far as needed, the application programmer may use the pointer provided by this function (typically named ctx) to set up any further, uncommon CMP client parameters. In this way we make sure that, as requested, the low-level API of CMPforOpenSSL defined in the C header file cmp.h, can be used jointly with this API.

One of the parameters of **CMPclient\_prepare()** is the callback function pointer transfer\_fn. By default (when NULL is given as the actual argument), standard HTTP(S) transfer is selected. In this case, a second function, **CMPclient\_setup\_HTTP()**, must be called next in order to provide the required HTTP parameters such as the server host and HTTP path.

Note that because CMP messages are self-contained any CMP implementation generally supports offline transport. Moreover, this implementation supports an overall timeout per CMP transaction and the default HTTP transfer supports a timeout per message exchange. Polling for requested certificates, as defined by [RFC4210], is fully supported and is done automatically when needed.

Then the actual activity for the given use case is invoked, by calling either **CMPclient\_imprint()**, **CMPclient\_bootstrap()**, **CMPclient\_update()**, or **CMPclient\_revoke()** with their use-case-specific arguments. Due to current technical limitations of the CMPforOpenSSL library, only one transaction can be performed with the same CMP context pointer (that is, it cannot be re-used for further transactions).

Finally the transaction must be closed by calling **CMPclient\_finish()**, which deallocates all internal resources in the given CMP context. In order to give advanced users the possibility to extract information from the CMP context before it is destroyed; this cleanup step has not been integrated at the end of the three above functions implementing the core use cases.

Thus, as the given level of abstraction, a typical invocation sequence would look like this:

**CMPErr** err = CMPclient\_init(log\_fn);

**CMP\_CTX** \*ctx = NULL;

…

err = CMPclient\_prepare(&ctx, log\_fn,

cmp\_truststore, recipient, untrusted,

creds, creds\_truststore, digest, mac,

transfer\_fn, total\_timeout,

new\_cert\_truststore, implicit\_confirm);

…

err = CMPclient\_setup\_HTTP(ctx, server, path, timeout, tls, proxy, no\_proxy);

…

err = CMPclient\_imprint(ctx, &new\_creds, new\_key, subject, exts);

…

CMPclient\_finish(ctx); /\* this also deallocates ctx \*/

The various parameters, as well as the meaning and the results of these functions are described in the next section.

Any number of transactions may be executed in a row or even in parallel as long as each of them uses its own CMP context pointer obtained by calling **CMPclient\_prepare()**.

The actual C header file with all relevant declarations can be found in the appendix.

The coding style of the library is compatible with the C90 standard.

# Core functionality

This section describes the essential functions of the generic CMP client library. These functions give feedback to the caller on their success or failure and the reason for any failure. We define the return type **CMP\_err** more abstractly than currently in the header file cmp.h of CMPforOpenSSL while the idea is the same: CMP\_OK = 0 (zero) means no error, else the code indicates the failure reason. The various error codes are defined in <openssl/cmperr.h>.

The function **CMPclient\_init()** initializes the underlying OpenSSL library and optionally sets up a log callback function as described in section 3.7 for use by the SecurityUtilities library.   
It should be called once, as soon as the overall application starts. If the log\_fn argument is NULL the library uses as default both the syslog facility and printing to the console.

**CMP\_err** **CMPclient\_init**(OPTIONAL **LOG\_cb\_t** log\_fn);

## CMPclient\_prepare

The function **CMPclient\_prepare()** allocates the internal CMP context data structure (of type **CMP\_CTX**) and set up those CMP parameters common to all use cases. On success, it assigns the pointer to the structure via the address of a variable that must be supplied as the first parameter. Note that this function, as well as the following ones, internally modify the CMP context and therefore this context is not declared **const**.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \*\* | pctx | Pointer to the variable that will obtain the context |
| OPTIONAL | **LOG\_cb\_t** | log\_fn | Function to be called for logging CMP related errors, warnings, etc. If NULL is given, **LOG\_default()** is used. See section 3.7 for details. |
| OPTIONAL | **X509\_STORE** \* | cmp\_ truststore | Trust store for authenticating the CMP server. For efficiency this data structure is not copied but its reference counter is incremented on success.  Although it might get modified, it may be reused. The argument may be NULL in case symmetric mutual authentication is done (via creds). |
| OPTIONAL | **const char** \* | recipient | X.509 Distinguished Name in the form "/<type0>=<value0>/<type1>=<value1>..." to use for the recipient field of CMP headers. If NULL then information from the creds or untrusted parameter or the NULL DN is taken as fallback. |
| OPTIONAL | **const STACK\_OF (X509)** \* | untrusted | Non-trusted intermediate CA certificates that may be useful for path construction when authenticating the server (i.e., the signer of received CMP response messages) and for verifying newly enrolled certificates. If NULL then any chain included in the creds parameter is taken as fallback. If the recipient argument is NULL and the creds argument is NULL or does not contain a client certificate the recipient of CMP messages sent is taken from the subject of the first certificate in this list, if any. |
| OPTIONAL | **const**  **CREDENTIALS** \* | creds | CMP client key material for protecting requests and authenticating to the server, or NULL in case requests should not be protected. Any password (symmetric key) included may also be used in opposite direction. See section 3.10 for details.  If a client certificate is included its subject is taken as the sender and, unless the recipient argument is given, its issuer is taken as fallback value for the recipient field of CMP messages.  If a certificate chain is included it is appended to the list of untrusted certificates. |
| OPTIONAL | **X509\_STORE** \* | creds\_ truststore | If this trust store is provided it is used to verify the chain building for the own CMP signer certificate. Otherwise an approximate chain is built as far as possible, ignoring errors. In both cases the untrusted certificates (with any creds->chain appended to them) are used for certificate path construction. |
| OPTIONAL | **const** **char** \* | digest | Name of hash function to use as one-way function (OWF) in PBM-based message protection and as digest algorithm for signature-based message protection and proof-of-possession (POPO) .  The default is "sha256".  The available digest names can be shown with the command openssl list -digest-commands |
| OPTIONAL | **const** **char** \* | mac | Name of MAC algorithm to use in RFC 4210’s MSG\_MAC\_ALG for PBM-based message protection.  The default is "hmac-sha1" as per RFC 4210. |
| OPTIONAL | **OSSL\_CMP\_transfer\_cb\_t** | transfer\_fn | Function to be called for message transfer.  See section 3.8 for details. |
|  | **int** | total\_ timeout | Maximum total time (in seconds) an enrollment (including polling) may take, or 0 for infinite, or < 0 for default, which is 0 (infinite) |
| OPTIONAL | **X509\_STORE** **\*** | new\_cert\_ truststore | Trust store to be used for verifying the newly enrolled certificate. See section 3.9 for details. For efficiency this data structure is not copied but its reference counter is incremented on success.  Although it might get modified, it may be reused. |
|  | **bool** | implicit\_ confirm | Flag whether to request implicit confirmation for enrolled certificates |

Example use (for the parts replaced by ‘…’, see sections 4.2 and 4.1):

**CMP\_err** err;

**CMP\_CTX** \*ctx = NULL;

**LOG\_cb\_t** log\_fn = NULL;

**X509\_STORE** \*cmp\_truststore = …

**const** **X509\_char\_\*recipient** = NULL;

**const** STACK\_OF(X509) \*untrusted = …

**CREDENTIALS** \*creds = …  
 **X509\_STORE** \*creds\_truststore = …

**const** **char** \*digest = "sha256";

**const** **char** \*mac = NULL;

**OSSL\_CMP\_transfer\_cb\_t** transfer\_fn = NULL; /\* default HTTP(S) transfer \*/

**int** total\_timeout = **100**;

**X509\_STORE** \*new\_cert\_truststore = …

**bool** implicit\_confirm = false;

err = CMPclient\_prepare(&ctx, log\_fn,

cmp\_truststore, recipient,

untrusted, creds, creds\_truststore, digest, mac,

transfer\_fn, total\_timeout,

new\_cert\_truststore, implicit\_confirm);

## CMPclient\_setup\_HTTP

The function **CMPclient\_setup\_HTTP()** sets up in the given CMP context the parameters relevant for HTTP transfer. As mentioned in section 2, this is only needed if HTTP(S) is used. All string parameters are copied and so may be deallocated immediately. The optional tls parameter is not copied, so must not be deallocated before invoking **CMPclient\_finish()**.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \* | ctx | CMP context to modify |
|  | **const** **char** \* | server | Server URI, of the form **"[http[s]://]<host>[:<port>][/path]",** with the default port being 80. Any given path is overridden by path if provided. |
| **OPTIONAL** | **const** **char** \* | path | Server HTTP path (aka CMP alias) with the default taken from the server argument, else “/”. |
|  | **int** | keep\_alive | If 0 then HTTP connections are closed after each response, which is the default for HTTP 1.0. If 1 or 2 is given then persistent connections within a transaction are requested. On 2 they are required, i.e., in case the server does not grant them an error occurs. A negative value assumes 1, which is the default for HTTP 1.1 and means preferring to keep the connection open. |
|  | **int** | timeout | Maximum time (in seconds) a single response to an HTTP POST request may take, or 0 for infinite, or < 0 for default, which is 120 seconds. |
| OPTIONAL | **SSL\_CTX** \* | tls | The TLS parameters if TLS shall be used, else NULL. For efficiency this data structure is not copied but its reference counter is incremented on success.  Although it might get modified, it may be reused. |
| OPTIONAL | **const** **char** \* | proxy | HTTP(S) proxy address, of the form  **"[http[s]://]<host>[:<port>][/path]"**,  with the default port being 80. Its default is the environment variable **http\_proxy or https\_proxy, respectively**. No proxy is used if the server host matches the no\_proxy setting. |
| OPTIONAL | **const** **char** \* | no\_proxy | List of server hosts not use an HTTP(S) proxy for, separated by commas and/or whitespace.  Default is the environment variable **no\_proxy**. |

Example use (for the parts replaced by ‘…’ and TLS\_new(), see section 4.4):

**const** **char** \*server = "ppki-playground.ct.siemens.com:443";

**const** **char** \*path = "/ejbca/publicweb/cmp/PlaygroundECC ";

**const** **char** \*proxy = NULL;

**const** **char** \*no\_proxy = NULL;

**int keep\_alive = -1;**

**int** timeout = **10**;

**X509\_STORE** \*tls\_truststore = …

**const** **STACK\_OF(X509)** \*tls\_untrusted = NULL;

**CREDENTIALS** \*tls\_creds = …

char \*ciphers = NULL; /\* or, e.g., "HIGH:!ADH:!LOW:!EXP:!MD5:@STRENGTH" \*/

**SSL\_CTX** \*tls = TLS\_new(tls\_truststore, tls\_untrusted,

tls\_creds, ciphers, -1);

err = CMPclient\_setup\_HTTP(ctx, server, path, keep\_alive, timeout, tls,

proxy, no\_proxy);

## CMPclient\_imprint, CMPclient\_bootstrap, and CMPclient\_pkcs10

The functions **CMPclient\_imprint()** and **CMPclient\_bootstrap()** perform a certificate enrollment, either an initial one (using the CMP command ‘**ir**’) or a regular one (using ‘**cr**’).

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \* | ctx | CMP context to use |
|  | **CREDENTIALS** \*\* | new\_creds | Pointer to variable to obtain the enrolled cert etc. |
|  | **const** **EVP\_PKEY** \* | new\_key | Private key for the new certificate; is is used for signature-based POPO, and the corresponding public key is put in the certificate template.  Note that an **EVP\_PKEY** structure can be used for both SW-based and HW-based keys. In the latter case it does not include the key material itself but a reference to key material via a crypto engine. |
|  | **const** **char** \* | subject | X.509 Subject Distinguished Name (DN) in the form "/<type0>=<value0>/<type1>=<value1>...". If the creds argument of is NULL or does not contain a certificate this name is taken as the sender field of the CMP messages sent. |
| OPTIONAL | **const** **X509\_EXTENSIONS** \* | exts | X.509 extensions to put in the cert template. |

The function **CMPclient\_pkcs10()** performs certificate enrollment based on a legacy PKCS#10 CSR using the CMP command ‘**p10cr**’.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \* | ctx | CMP context to use |
|  | **CREDENTIALS** \*\* | new\_creds | Pointer to variable to obtain the enrolled cert etc. |
|  | **const** **X509\_REQ** \* | pc10csr | Legacy PKCS#10 certificate signing request to use |

On success, each of the enrollment functions allocates a **CREDENTIALS** structure and fills it with the new\_key argument supplied (or NULL in case of **CMPclient\_pkcs10()**), the newly enrolled certificate, and a chain for this certificate. The chain is constructed from the list of untrusted certificates held in the CMP context, which includes any certificates provided by the server in the extraCerts field of responses. The pointer to the structure is returned via the pointer to a variable supplied as the new\_creds parameter.

Example use (for KEY\_new(), see section 4.3, for the part replaced by ‘…’, see section 4.5, and for CREDENTIALS\_save(), see section 4.1):

**CREDENTIALS** \*new\_creds = NULL;

**const** **char** \*subject = "/CN=test-genCMPClient/OU=PPKI Playground/OU=Corporate Technology"  
 "/OU=For internal test purposes only/O=Siemens/C=DE";

**EVP\_PKEY** \*new\_key = KEY\_new("EC:prime256v1");

**X509\_EXTENSIONS** \*exts = …

err = CMPclient\_bootstrap(ctx, &new\_creds, new\_key, subject, exts);

**if** (err == 0) {

**const** **CREDENTIALS** \*creds = new\_creds;

**const** **char** \*file = "certs/new.p12";

**const** **char** \*source = NULL /\* plain file \*/;

**const** **char** \*desc = " newly enrolled certificate and related key and chain";

**if** (!CREDENTIALS\_save(creds, file, file, OPTIONAL source, OPTIONAL desc))

**goto** err;

}

All enrollment functions described in this section as well as **CMPclient\_update()** described in the next section are implemented internally via a combination of the functions **CMPclient\_** **setup\_certreq()** and **CMPclient\_enroll()**. For more flexibility these may be called directly.

## CMPclient\_update

The function **CMPclient\_update()** performs a certificate update, aka re-enrollment using the CMP command ‘**kur**’. The certificate to be updated is the cert component of the creds argument given to **CMPclient\_prepare()**.On success, a **CREDENTIALS** structure is returned as described above in section 3.3 for the enrollment functions.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \* | ctx | CMP context to use |
|  | **CREDENTIALS** \*\* | new\_creds | Pointer to variable to obtain the enrolled cert etc. |
|  | **const** **EVP\_PKEY** \* | new\_key | Key (pair); see above description in section 3.3 |

## CMPclient\_revoke

The function **CMPclient\_revoke()** performs certificate revocation using the CMP command ‘**rr**’.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \* | ctx | CMP context to use |
|  | **const** **X509** \* | cert | Certificate to be revoked |
|  | **int** | reason | Revocation reason code, as defined in openssl/x509v3.h |

## CMPclient\_finish

The function **CMPclient\_finish()** deallocates the given CMP context, deallocating all internal data but not the structures passed in via the functions described before. Due to [current limitations of CMPforOpenSSL](https://github.com/mpeylo/cmpossl/issues/101) [cmpossl], only one invocation of the functions described in 3.3, 3.4, and 3.5 can be done with the same context structure. Any of the pointers provided for the above truststore, creds, server, path, tls, subject, newkey, exts, or cert parameters can be reused by the caller and must be deallocated when not needed any more.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
|  | **CMP\_CTX** \* | ctx | CMP context to deallocate |

Example use (for the various \_free() functions, see sections 3.10 and 4):

CMPclient\_finish(ctx);

CREDENTIALS\_free(new\_creds);

EXTENSIONS\_free(exts);

KEY\_free(newkey);

TLS\_free(tls);

CREDENTIALS\_free(tls\_creds);

STORE\_free(tls\_truststore);

STORE\_free(new\_cert\_truststore);  
 STORE\_free(cmp\_truststore);  
 CREDENTIALS\_free(creds);  
 LOG\_close();

## Logging

When an important activity is performed or an error occurs, some more detail should be provided for debugging and auditing purposes. An application can obtain this information by providing a callback function, which is called on error with component, file, lineno, and msg arguments that may provide a component identifier, a file path name and a line number indicating the source code location and a string describing the nature of the event.

Even when an activity is successful some warnings may be useful and some degree of logging may be required. Therefore we have extended the type of the logging callback function of CMPforOpenSSL by a level argument indicating the severity level, such that error, warning, info, debug, etc. can be treated differently. Moreover, the callback function may itself do non-trivial tasks like writing to a log file, which in turn may fail. Thus we utilize a Boolean return type indicating success or failure.

**typedef enum { false = 0, true = 1 } bool;  
typedef** **int** **severity**;

#define LOG\_EMERG 0

#define LOG\_ALERT 1

#define LOG\_CRIT 2

#define LOG\_ERR 3

#define LOG\_WARNING 4

#define LOG\_NOTICE 5

#define LOG\_INFO 6

#define LOG\_DEBUG 7

#define LOG\_TRACE 8

**typedef** **bool** (\***LOG\_cb\_t**) (OPTIONAL **const** **char** \*func,

OPTIONAL **const** **char** \*file, **int** lineno,

**severity** level, **const** **char** \*msg);

The **LOG\_default()** function prints more critical messages like errors and warnings to stderr, while less critical ones like info and debug messages are printed to stdout. In addition, it sends messages to syslog.

**bool** **LOG\_default**(OPTIONAL **const** **char** \*func, OPTIONAL **const** **char** \*file, **int** lineno,

**severity** level, **const** **char** \*msg);

The **LOG\_set\_verbosity()** function sets sets the verbosity of **LOG\_default()**. The level parameter defines the minimal severity of messages to be output. The default is LOG\_INFO.

**void** **LOG\_set\_verbosity**(**severity** level);

The **LOG\_set\_name()** function sets the application name printed by the default log output. The string pointed to by the name parameter must not be deallocated as long as logging is used. The default is "SecUtils" .

**void** **LOG\_set\_name**(OPTIONAL **const** **char** \*name);

The **LOG()** function logs the message specified by the parameter fmt and optionally further ones as with **printf()**. The parameter func optionally gives the name of the reporting function or component, the parameter file optionally gives the the current source file path name, the parameter lineno gives the current line number or 0, and the parameter level gives the nature of the message. The function returns true success and false on failure.

**bool** **LOG**(OPTIONAL **const** **char** \*func, OPTIONAL **const** **char** \*file, **int** lineno,

**severity** level, **const** **char** \*fmt, ...);

When all CMP client activity is finished the log should be closed using the following function, which flushes any pending log output and deallocates log-related resources.

**void** LOG\_close(**void**);

## Message transfer callback function

The usual way of transferring CMP messages is via HTTP (see also [RFC6712]), with or without TLS. As mentioned in section 2, this transfer mode is therefore the default. Yet it is possible to provide as the transfer\_fn argument of the **CMPclient\_prepare()** function (see section 3.1) a non-NULL function pointer of type **OSSL\_CMP\_transfer\_cb\_t**. This callback function takes as parameters the current CMP context structure, the request message to be sent and the address of a result variable to which it shall assign on success the response message received at the end of the transfer. The function shall send the request to some server and try to obtain the corresponding response from the server. It shall return an error code of type **CMP\_err**. If needed, the application may also provide a further argument to the callback function, using the CMPforOpenSSL functions **OSSL\_CMP\_CTX\_set\_transfer\_cb\_arg()** and **OSSL\_CMP\_CTX\_get\_transfer\_cb\_arg()**. This API design gives full freedom for implementing whatever method of transferring methods, including file-based ones.

**typedef** **OSSL\_CMP\_MSG** \*(\***OSSL\_CMP\_transfer\_cb\_t**)(**CMP\_CTX** \*ctx,  
 **const** **OSSL\_CMP\_MSG** \*req);

## Certificate checking callback function

When the CMP client receives from the server a newly enrolled certificate it should have the possibility to inspect the certificate to check whether it fulfills the given expectations. Depending on the outcome of this check, the client can signal acceptance or rejection of the certificate to the server via the ‘certConf’ CMP message.

The CMPforOpenSSL library just checks that the public key in the new certificate matches the key used in the request. In addition one can provide via the **OSSL\_CMP\_CTX\_set\_certConf\_cb()** function a function pointer of type **OSSL\_CMP\_certConf\_cb\_t**. This callback function takes as parameters the current CMP context structure, the newly enrolled certificate to be checked, any CMP failure bits (see [RFC4210, section 5.2.3]) already determined by the library, and a pointer to the string result variable to which it may assign on error a string describing why it rejects the given certificate. The function shall return 0 on acceptance or CMP failure bits with indices between 0 and OSSL\_CMP\_PKIFAILUREINFO\_MAX (= 26) indicating the reason(s) for rejection. The application may also provide a further, implicit argument to the callback function via the CMPforOpenSSL function **OSSL\_CMP\_CTX\_set\_certConf\_cb arg()**. This argument can be retrieved using **OSSL\_CMP\_CTX\_get\_certConf\_cb\_arg()**.

If the new\_cert\_truststore argument of the **CMPclient\_prepare()** is not NULL the callback function **OSSL\_CMP\_certConf\_cb()** provided by CMPforOpenSSL will be selected, which uses this argument as a trust store for validating the newly enrolled certificate.

This API design gives full freedom for implementing arbitrary checks on newly enrolled certificates, for instance whether the subject DN is as expected and/or all required X.509 extensions have been set, in addition to validating the certificate relative to some trust store.

**typedef** **int** (\***OSSL\_CMP\_certConf\_cb\_t**)(CMP\_CTX \*ctx, **const** X509 \*cert,

**int** fail\_info, **const** **char** \*\*txt);

## Component credentials

Like CMPforOpenSSL, for key material and other core crypto data structures we re-use the ones defined by the underlying OpenSSL library, as far as possible, but one was missing.

It is very useful to have an abstraction that combines the key material a component has for authenticating itself in a single data structure. For signature-based authentication this consists of a private key (of OpenSSL type **EVP\_PKEY**, which can refer to a key held in a hardware key store via a crypto engine), the current certificate (of OpenSSL type **X509**) including the corresponding public key, and optionally the chain of its issuer certificates towards the respective root CA (of OpenSSL type **STACK\_OF(X509)).** For authentication with password-based MAC (PBM) the credentials include the password and optionally a reference value that may be needed, similarly to a user name, to identify which password to use.

The resulting data structure, which we call **CREDENTIALS**, will be used by the CMP client on the one hand for itself, namely for signing/protecting CMP messages and optionally for authenticating itself as TLS client, and on the other hand to convey the output of certificate enrollment, where the newly enrolled certificate is bundled with the related private key and any chain of certificates provided by the server.

**typedef** **struct** **credentials** {

OPTIONAL **EVP\_PKEY** \*pkey;

OPTIONAL **X509** \*cert;

OPTIONAL **STACK\_OF(X509)** \*chain;

OPTIONAL char \*pwd;

OPTIONAL char \*pwdref;

} **CREDENTIALS**;

We define the following core functions dealing with credentials.

The function **CREDENTIALS\_new()** constructs a set of credentials from its components (i.e., a private key, a related certificate, and optionally a chain, and/or a password and optionally its reference value) and returns on a pointer to the newly allocated structure on success or NULL on failure (i.e., out of memory). On success the reference counter of the first three arguments are incremented and the last two arguments are copied. This means that the caller can deallocate all provided arguments immediately and in any case should wipe/erase the contents of the pwd parameter right away for security reasons.

|  |  |  |  |
| --- | --- | --- | --- |
| Param. | Type | Name | Meaning |
| OPTIONAL | **const** **EVP\_PKEY** \* | pkey | Private key to include, which may be software-based or stored in an engine |
| OPTIONAL | **const** **X509** \* | cert | Related certificate to include |
| OPTIONAL | **const** **STACK\_OF(X509)** \* | chain | Chain of the given certificate |
| OPTIONAL | **const** **char \*** | pwd | Password to use for PBM etc. |
| OPTIONAL | **const** **char \*** | pwdref | Reference for identifying the password |

The function **CREDENTIALS\_free()** takes a pointer to a credentials structure when not needed any more, deallocates its components (using among others **KEY\_free()**, which wipes the private key, and **OPENSSL\_cleanse()** to wipe the secret/password), and then deallocates the structure itself. It has no return value.

**void CREDENTIALS\_free**(OPTIONAL **CREDENTIALS** \*creds);

Since the **CREDENTIALS** data type is opaque some selector functions are needed:

**X509** \***CREDENTIALS\_get\_cert**(**const** **CREDENTIALS** \*creds);

**STACK\_OF(X509)** \***CREDENTIALS\_get\_chain**(**const** **CREDENTIALS** \*creds);

# Support functionality

This section describes useful auxiliary functions for preparing the parameters of the above core functions. While this could be done directly using the rather low-level OpenSSL API, it is cumbersome and error-prone to identify and directly use the OpenSSL functions directly. Therefore we introduce this intermediate level for convenience, such that the typical use cases can be implemented without needing to know any details of the underlying CMPforOpenSSL and OpenSSL API. As mentioned before, experienced programmers may still make use of those lower-level functions in order to cover any special needs.

## CREDENTIALS helpers

Since certificates as well as private keys (unless they are held in a hardware key store) are usually held in files, we provide functions for loading the components of credentials from files and for saving them in files. For now, we focus here on the PKCS#12 file format because all components of a **CREDENTIALS** structure can be easily and conveniently managed in a single PKCS#12 structure. Other formats, such as PEM, are partially supported.

The function **CREDENTIALS\_load()** reads from the file given in the certs argument (if not NULL) the primary certificate, which is taken as the cert component, plus any further ones, which are taken as the chain component. In case the source argument is NULL or begins with “pass:”, it reads the private key from the file given in the key argument (if not NULL), where the source argument may refer to a password in the form “pass:<pwd>” that may be needed to decrypt the file contents including the private key. If the certs and key arguments are equal the credentials are jointly read from the same file, which is expected in PKCS#12 format, else for each file the format may be PEM, PKCS#12, or ASN.1 (DER). In case the source argument begins with “engine:”, it loads a reference to the private key with the identifier given in the key argument, where the rest of the source argument gives the identifier of the crypto engine to use (while the remaining credentials components are loaded from the file without decrypting it). The respective crypto engine must already have been parameterized and initialized in an engine-specific way with the usual OpenSSL mechanisms, which are described for instance in .

The function internally calls **CREDENTIALS\_new()** to construct a **CREDENTIALS** structure and returns the pointer to it on success, or NULL otherwise. In case of errors optionally the string held in the optional desc parameter is used for forming more descriptive error messages.

**CREDENTIALS** \***CREDENTIALS\_load**(OPTIONAL **const** **char** \*certs, OPTIONAL **const** **char** \*key,

OPTIONAL **const** **char** \*source, OPTIONAL **const** **char** \*desc);

Example use for certificates and a private key read from a PKCS#12 file:

**const** **char** \*certs = "certs/ppki\_playground\_cmp\_signer.p12";

**const** **char** \*key = certs;

**const** **char** \*source = "pass:12345";

**const** **char** \*desc = "credentials for CMP level"

**CREDENTIALS** \*creds = CREDENTIALS\_load(certs, key, source, desc);

Example use where the private key is held in HW and its reference is loaded via PKCS#11 (while the actual key is held, e.g., on a smart card or a TPM chip):

**const** **char** \*tls\_certs = "certs/ppki\_playground\_tls.p12";

**const** **char** \*tls\_pkey = "my-key-ID;type=private;pin-value=1234";

**const** **char** \*tls\_source = "engine:pkcs11";

**const** **char** \*tls\_desc = "credentials for TLS level"

**CREDENTIALS** \*tls\_creds = CREDENTIALS\_load(tls\_certs, tls\_pkey, tls\_source,

tls\_desc);

The function **CREDENTIALS\_save()** writes the certificate components of the given credentials data structure creds to the file given as the certs argument (unless it is null). If the certs and key arguments are equal the certificates and the private key are written jointly to the same PKCS#12 file, else they are written to PEM files (where the certificates are not encrypted). In case the source argument is NULL or begins with “pass:”, it stores the private key in the given key file (unless it is null). , where the source argument may refer to a password in the form “pass:<pwd>” that is then used to encrypt the private key (together with the related certificates when stored jointly in a PKCS#12 file) before storing it. In case the source argument begins with “engine:”, it assumes that the private key is held in a crypto engine and there is no need and neither a possibility for it to save the key (nor to encrypt the related certificates written to a file). The function returns true on success and false otherwise. In case of errors optionally the string held in the optional desc parameter is used for forming error messages.

**bool** **CREDENTIALS\_save**(OPTIONAL **const** **CREDENTIALS** \*creds,

OPTIONAL **const** **char** \*certs, OPTIONAL **const** **char** \*key,

OPTIONAL **const** **char** \*source, OPTIONAL **const** **char** \*desc);

An example use has already been given in section 3.3.

There are also functions for loading and storing individual certificates and for loading CSRs:

**X509** \***CERT\_load**(**const** **char** \*file, OPTIONAL **const** **char** \*pass, OPTIONAL **const** **char** \*desc);

**bool** **CERT\_save**(**const** **X509** \*cert, **const** **char** \*file, OPTIONAL **const** **char** \*desc);

**X509\_REQ** \***CSR\_load**(**const** **char** \*file, OPTIONAL **const** **char** \*desc);

When storing a certificate in a file with the given name the format is determined from the file name extension and can be PEM, DER, or PKCS#12. In case of errors the string held in the optional desc parameter is used for forming more descriptive error messages.

Functions for loading and saving lists of certificates are described below.

## ***X509\_***STORE helpers

As the above core functions reuse the OpenSSL trust store data structure of type **X509\_STORE** and such a structure is non-trivial to manage we provide helper functions for this purpose. For instance, the store needs to be initialized with trusted certificates and optionally with many other verification parameters such as Certificate Revocation Lists (CRLs), URLs of Certificate Distribution Points (CDPs), and Online Certificate Status Protocol (OCSP) responders. Certificates are typically held in files and thus need to be loaded while CRLs are typically retrieved online from CDPs and then cached in files or in memory. See also our general certificate validation guideline [Cert\_Valid].

The function **STORE\_load()** sets up a new trust store with the certificates held in the PEM, DER, or PKCS#12 file(s) with the comma-separated list of names given as the trusted\_certs argument. It enables diagnostic output in the log that is very helpful for debugging in case certificate verification fails. It does not enable certificate status checks. The function returns the pointer to the constructed trust store on success, or NULL otherwise. In case of errors the string held in the optional desc parameter is used for forming error messages.

**X509\_STORE** \***STORE\_load**(**const** **char** \*trusted\_certs, OPTIONAL **const** **char** \*desc);

Example use:

**const** **char** \*trusted\_certs = "certs/trusted/PPKIPlaygroundECCRootCAv10.crt,"

"certs/trusted/PPKIPlaygroundInfrastructureRootCAv10.crt";

**const** **char** \*desc = "trusted certs for CMP level";  
 **X509\_STORE** \*truststore = STORE\_load(trusted\_certs, OPTIONAL desc);

The function **CERTS\_load()** loads the certificate(s) held in the PEM, DER, or PKCS#12 file(s) with the comma-separated list of file names in the files argument and returns the pointer to the loaded list of certificates on success, or NULL otherwise. These certificates can be used as auxiliary untrusted certs when constructing a **CMP\_CTX** or **TLS\_CTX**. In case of errors the string held in the optional desc parameter is used for forming more descriptive error messages.

**STACK\_OF(X509)** \***CERTS\_load**(**const** **char** \*files, OPTIONAL **const** **char** \*desc);

The function **CERTS\_save()** stores the given certificate(s) in a file with the given name, where the format is determined from the file name extension and can be PEM, DER, or PKCS#12. In case of errors the string held in the optional desc parameter is used for forming more descriptive error messages.

**int** **CERTS\_save**(**const** **STACK\_OF(X509)** \*certs, **const** **char** \*file, OPTIONAL **const** **char** \*desc);

The function **CERTS\_free()** deallocates any given list of certificates. It has no return value.

**void** **CERTS\_free**(OPTIONAL **STACK\_OF(X509)** \*certs);

The function **CRLs\_load()** loads the CRL(s) held in the DER or PEM file(s) with the comma-separated list of file names in the files argument and returns the loaded list of CRLs on success, or NULL otherwise. The timeout parameter specifies the number of seconds an HTTP transaction (if needed) may take, or 0 for infinite or -1 for default. In case of errors the string held in the optional desc parameter is used for forming more descriptive error messages.

**STACK\_OF(X509\_CRL)** \*CRLs\_load(**const** **char** \*files, **int** timeout, OPTIONAL **const** **char** \*desc);

The function **STORE\_add\_crls**() adds an optional list of CRLs to the given trust store and enables CRL-based status checks for end-entity certificates.

**bool** **STORE\_add\_crls**(**X509\_STORE** \*truststore, OPTIONAL **const** **STACK\_OF(X509\_CRL)** \*crls);

The function **CRLs\_free()** deallocates any given list of CRLs. It has no return value.

**void** **CRLs\_free**(OPTIONAL **STACK\_OF(X509\_CRL)** \*crls);

The function **STORE\_set\_parameters()** sets various optional verification parameters in the given trust store truststore; in more detail, it

* takes over any given OpenSSL certificate verification parameters vpm
* demands certificate status checks in case any of the OCSP- or CRL-related options is set. If in addition the full\_chain option is set then all (except root) certificates are checked, else only end-entity certificates, i.e., the first certificate of each chain.   
  For each certificate for which the status check is demanded the verification function will try to obtain the revocation status first via OCSP stapling (which is applicable only for TLS) if enabled, then from any locally available CRLs, then from any OCSP responders if enabled, and finally from any certificate distribution points (CDPs) if enabled.   
  Verification fails if no valid and current revocation status can be determined or the status indicates that the certificate has been revoked.
* enables OCSP stapling, which makes sense only for TLS, if try\_stapling is set
* adds any CRLs provided in the crls argument and in this case enables CRL-based checks,
* enables CRL-based checks in case the use of CDP entries in certificates is enabled via the use\_CDP argument or a static list of comma-separated URLs for fetching CRLs is given as the cdps argument (which is used as fallback) where the crls\_timeout parameter gives the number of seconds fetching a CRL may take, or 0 for infinite or -1 for default (= 10), and
* enables fetching OCSP responses in case the use of AIA OCSP entries in certificates is enabled via the use\_AIA argument or a static list of comma-separated OCSP responder URLs is given as the ocsp argument (which is used as fallback) where the ocsp\_timeout parameter gives the number of seconds fetching an OCSP response may take, or 0 for infinite or -1 for default (= 10).

The function returns true on success and false otherwise.

**bool** **STORE\_set\_parameters**(**X509\_STORE** \*truststore, OPTIONAL **const** **X509\_VERIFY\_PARAM** \*vpm,

**bool** full\_chain, **bool** try\_stapling,

OPTIONAL **const** **STACK\_OF(X509\_CRL)** \*crls,

**bool** use\_CDP, OPTIONAL **const** **char** \*cdps, **int** crls\_timeout,

**bool** use\_AIA, OPTIONAL **const** **char** \*ocsp, **int** ocsp\_timeout);

The function **STORE\_set\_crl\_callback()** sets a CRL fetching callback function and optional argument in the given trust store, which may be used for instance to implement CRL caching. If the use\_CDP parameter has been set in the trust store the callback is called for each HTTP URL found in the CDP entries of a certificate to be verified. If all these are inconclusive then it is called once more with a NULL URL (such that the callback may try getting a CRL based on any further information contained in the certificate being checked). The store parameter references the certificate trust store to be extended, the crl\_cb parameter provides the callback function to use, or null for default (which is **CONN\_load\_crl\_http()**), and the crl\_cb\_arg may be used to provide an argument to be passed to the callback function. It returns true on success and false otherwise.

**typedef** **X509\_CRL** \*(\***CONN\_load\_crl\_cb\_t**)(OPTIONAL **void** \*arg,

OPTIONAL **const** **char** \*url, **int** timeout,

**const** **X509** \*cert, OPTIONAL **const** **char** \*desc);

**bool** **STORE\_set\_crl\_callback**(**X509\_STORE** \*store,

OPTIONAL **CONN\_load\_crl\_cb\_t** crl\_cb,

OPTIONAL **void** \*crl\_cb\_arg);

The **STORE\_set1\_host\_ip()** function enables host verification in the given trust store (typically returned from **TLS\_new()**) and defines the server host name and/or IP address to be expected for a TLS connection. It it is crucial for TLS clients to verify the identity of the host to connect to. The parameter host optionally gives the host DNS name to be expected, while the parameter ip optionally gives the host IP address to be expected. If both host and ip are non-NULL and are equal the function tries to interpret the string first as IP address then as domain name. The strings are copied (until any ':' is found, i.e., any port specification is ignored). The function returns true on success and false otherwise..

**bool** **STORE\_set1\_host\_ip**(**X509\_STORE** \*store, **const** **char** \*host, **const** **char** \*ip);

Further non-default trust store parameters may be set as far as needed using the various respective low-level OpenSSL functions.  
Example for setting up a trust store with use of statically and dynamically obtained CRLs:

**X509\_STORE** \*truststore = …;

**const** **X509\_VERIFY\_PARAM** \*vpm = NULL;

**bool** full\_chain = true;

**bool** try\_stapling = false;

**const** **char** \*file =

"certs/crls/PPKIPlaygroundInfrastructureIssuingCAv10.crl, "

"certs/crls/PKIPlaygroundECCRootAv10.crl";

**const** **char** \*desc = "CRLs for CMP level";

**const** **STACK\_OF(X509\_CRL)** \*crls = CRLs\_load(file, -1, NULL);

**bool** use\_CDP = true;

**const** **char** \*cdps = NULL;

**int** crls\_timeout = -1;

**bool** use\_AIA = false;

**const** **char** \*ocsp = NULL;

**int** ocsp\_timeout = -1;

**bool** success = STORE\_set\_parameters(truststore, vpm, full\_chain,

try\_stapling, crls,

use\_CDP, cdps, crls\_timeout,  
 use\_AIA, ocsp, ocsp\_timeout);

The function **STORE\_free()** deallocates any given trust store. It has no return value.

**void** **STORE\_free**(OPTIONAL **X509\_STORE** \*truststore);

## EVP\_PKEY helpers

The function **KEY\_new()** generates a new private key of OpenSSL type **EVP\_PKEY** according to the specification given as its spec argument, which may be of the form “RSA:<length>” or “EC:<curve>”. The RSA key length must be between 1024 and 8192 bits.   
The available ECC curves can be shown with the command openssl ecparam -list\_curves.   
The function returns the new key on success and NULL otherwise.

**EVP\_PKEY** \***KEY\_new**(**const** **char** \*spec);

An example use has been given in section 3.3.

Keys accessed via a crypto engine need to be generated by other (engine-specific) means.

The **KEY\_load()** function loads a private key from the given file or engine. The file format can be PEM, DER, or PKCS#12. The parameter pass may provide a password (optionally preceded by "pass:") needed for decrypting the file content. In case of errors the string held in the optional desc parameter is used for forming more descriptive error messages.

**EVP\_PKEY** \***KEY\_load**(OPTIONAL **const** **char** \*file, OPTIONAL **const** **char** \*pass,

OPTIONAL **const** **char** \*engine, OPTIONAL **const** **char** \*desc);

The function **KEY\_free()** deallocates the given pkey and wipes its representation in memory if it is SW-based. It has no return value. For HW-based keys it just deallocates the reference.

**void** **KEY\_free**(OPTIONAL **EVP\_PKEY** \*pkey);

## SSL\_CTX helpers

The function **TLS\_new()** sets up a new OpenSSL **SSL\_CTX** structure with reasonable default parameters for TLS (typically HTTPS client) connections. See also the Siemens TLS configuration [TLS\_Conf] and integration guidelines [TLS\_Integration]. Its optional arguments are the trust store truststore to use for authenticating the peer (typically a TLS server), a list of intermediate certificates untrusted that may be helpful when building the own (typically TLS client) certificate chain and while checking stapled OCSP responses, the credentials creds to use for authenticating to the peer, and the enabled cipher suites. All these parameters are not consumed, so should be deallocated by the caller.

If the creds argument is given the function checks that its certificate matches its private key and tries to build a chain from the certificate using the truststore and untrusted certificates (as far as given) to be used for authenticating to the peer. If this fails it issues a warning and uses the chain in the creds argument (if included) as fallback.

The available cipher suite names can be shown with the command openssl list -cipher-algorithms. See also [OpenSSL-ciphers] how to specify them more abstractly.   
The security level ranges from 0 (lowest) to 5. If -1 is given, a sensible value is determined from the cipher list if provided, else the OpenSSL default is used (which is currently 1). For details see [OpenSSL-sec-level]. The function returns the pointer to the new structure on success, or NULL otherwise. Further non-default TLS parameters may be set as far as needed using the various respective low-level OpenSSL functions.

**SSL\_CTX** \***TLS\_new**(OPTIONAL **X509\_STORE** \*truststore,

OPTIONAL **const** **STACK\_OF(X509)** \*untrusted,  
 OPTIONAL **const** **CREDENTIALS** \*creds,

OPTIONAL **const** char \*ciphers, **int** security\_level);

The function **TLS\_free()** deallocates the given TLS context tls . It has no return value.

**void** **TLS\_free**(OPTIONAL **SSL\_CTX** \*tls);

## X509\_EXTENSIONS helpers

The function **EXTENSIONS\_new()** initiates a list of X.509 extensions, which has OpenSSL type **X509\_EXTENSIONS**, to be used in certificate enrollment. It returns the pointer to the new structure on success, or NULL otherwise.

**X509\_EXTENSIONS** \***EXTENSIONS\_new**(**void**);

The function **EXTENSIONS\_add\_SANs()** appends to the given list of X.509 extensions exts a list of Subject Alternative Names (SANs) given as a string spec of comma-separated domain names, IP addresses, and/or URIs optionally preceded by “critical,” to mark them critical. It returns true on success and false otherwise.

**bool** **EXTENSIONS\_add\_SANs**(**X509\_EXTENSIONS** \*exts, **const** **char** \*spec);

The function **EXTENSIONS\_add\_ext()** appends to the given list of X.509 extensions exts an extension of the given type, e.g., "basicContraints", "keyUsage", "extendedKeyUsage", or   
"certificatePolicies". Its value is given as a string spec of comma-separated names or OIDs optionally preceded by “critical,” to mark the extension critical. The specification may refer to further details specified in the style of OpenSSL configuration file sections (see [OpenSSL-config]), which can be provided via the optional sections parameter. The function returns true on success, false otherwise.

Possible values for basic key usages are: "digitalSignature", "nonRepudiation",

"keyEncipherment", "dataEncipherment", "keyAgreement", "keyCertSign", "cRLSign", "encipherOnly", and "decipherOnly". For a list of generally defined Extended Key Usage OIDs, see [OID-ref].

**bool** **EXTENSIONS\_add\_ext**(**X509\_EXTENSIONS** \*exts, **const** **char** \*name, **const** **char** \*spec, **BIO** \*sections);

Example use:

**X509\_EXTENSIONS** \*exts = EXTENSIONS\_new();

**BIO** \*policy\_sections = BIO\_new(BIO\_s\_mem());

**bool** success = exts != NULL && policy\_sections != NULL &&

EXTENSIONS\_add\_SANs(exts, "localhost, 127.0.0.1, 192.168.0.1") &&  
 EXTENSIONS\_add\_ext (exts, "keyUsage", "critical, digitalSignature", NULL) &&

EXTENSIONS\_add\_ext (exts, "extendedKeyUsage", "critical, clientAuth, "  
 "1.3.6.1.5.5.7.3.1" /\* serverAuth \*/, NULL) &&  
 BIO\_printf(policy\_sections, "%s",   
 "[pkiPolicy]\n"  
 " policyIdentifier = 1.3.6.1.4.1.4329.38.4.2.2\n"  
 " CPS.1 = http://www.siemens.com/pki-policy/\n"  
 " userNotice = @notice\n"  
 "[notice]\n"  
 " explicitText=Siemens policy text\n") > 0 &&  
 EXTENSIONS\_add\_ext(exts, "certificatePolicies",  
 "critical, @pkiPolicy", policy\_sections);  
 BIO\_free(policy\_sections);

The function **EXTENSIONS\_free()** deallocates the given structure exts. It has no return value.

**void** **EXTENSIONS\_free**(OPTIONAL **X509\_EXTENSIONS** \*exts);

# Appendix: C header file

/\*!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file genericCMPclient.h

\* @brief generic CMP client library API

\*

\* @author David von Oheimb, Siemens T RDA CST SEA, [David.von.Oheimb@siemens.com](mailto:David.von.Oheimb@siemens.com)

\*

\* Copyright (c) Siemens AG, 2018-2021.

\* Licensed under the Apache License, Version 2.0

\* SPDX-License-Identifier: Apache-2.0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef GENERIC\_CMP\_CLIENT\_H

# define GENERIC\_CMP\_CLIENT\_H

/\* for low-level CMP API, in particular, type **CMP\_CTX** \*/

# include <openssl/cmp.h>  
  
/\* for abbreviation and backward compatibility: \*/

**typedef** **OSSL\_CMP\_CTX CMP\_CTX**;   
**typedef** **OSSL\_CMP\_severity severity**;

**typedef** **int** **CMP\_err**; /\* should better be defined and used in openssl/cmp.h \*/

# define CMP\_OK 0

# define CMP\_R\_LOAD\_CERTS 255

# define CMP\_R\_LOAD\_CREDS 254

# define CMP\_R\_GENERATE\_KEY 253

# define CMP\_R\_STORE\_CREDS 252

# define CMP\_R\_RECIPIENT 251

# define CMP\_R\_INVALID\_CONTEXT 250

# define CMP\_R\_INVALID\_PARAMETERS 249/\* further error codes are defined in ../cmpossl/include/openssl/cmperr.h \*/

# define CMP\_IR 0

# define CMP\_CR 2

# define CMP\_P10CR 4

# define CMP\_KUR 7

# define CMP\_RR 11

# ifndef \_\_cplusplus

**typedef** **enum** { false = 0, true = 1 } **bool**; /\* Boolean value \*/

# endif

#define OPTIONAL /\* marker for non-required parameter, i.e., NULL allowed \*/

/\* private key and related certificate, plus optional chain \*/

**typedef** **struct** **credentials** {

OPTIONAL **EVP\_PKEY** \*pkey; /\* can refer to HW key store via engine \*/

OPTIONAL **X509** \*cert; /\* related certificate \*/

OPTIONAL **STACK\_OF(X509)** \*chain; /\* intermediate/extra certs for cert \*/

OPTIONAL **const** **char** \*pwd; /\* alternative: password (shared secret) \*/

OPTIONAL **const** **char** \*pwdref; /\* reference identifying the password \*/  
} **CREDENTIALS**;

**typedef int** **severity**;

# define LOG\_EMERG 0

# define LOG\_ALERT 1

# define LOG\_CRIT 2

# define LOG\_ERR 3

# define LOG\_WARNING 4

# define LOG\_NOTICE 5

# define LOG\_INFO 6

# define LOG\_DEBUG 7

# defineLOG\_TRACE 8

typedef **bool** (\***LOG\_cb\_t**) (OPTIONAL **const** **char** \*func,

OPTIONAL **const** **char** \*file, **int** lineno,

**severity** level, **const** **char** \*msg);

/\* CMP client core functions \*/  
/\* should be called once, as soon as the application starts \*/

**CMP\_err** **CMPclient\_init**(OPTIONAL **LOG\_cb\_t** log\_fn);

/\* must be called first \*/

**CMP\_err** **CMPclient\_prepare**(**CMP\_CTX** \*\*pctx,   
 OPTIONAL **LOG\_cb\_t** log\_fn,

OPTIONAL **X509\_STORE** \*cmp\_truststore,

OPTIONAL **const** **char** \*recipient,

OPTIONAL **const** **STACK\_OF(X509)** \*untrusted,

OPTIONAL **const** **CREDENTIALS** \*creds,

OPTIONAL **X509\_STORE** \*creds\_truststore,

OPTIONAL **const** **char** \*digest,

OPTIONAL **const** **char** \*mac,

OPTIONAL **OSSL\_CMP\_transfer\_cb\_t** transfer\_fn, **int** total\_timeout,

OPTIONAL **X509\_STORE** \*new\_cert\_truststore, **bool** implicit\_confirm);

/\* must be called next in case the transfer\_fn is NULL, which implies HTTP\_transfer \*/

**CMP\_err** **CMPclient\_setup\_HTTP**(**CMP\_CTX** \*ctx, **const** **char** \*server, **const** **char** \*path,

**int** timeout, OPTIONAL **SSL\_CTX** \*tls,

OPTIONAL **const** **char** \*proxy, OPTIONAL **const** **char** \*no\_proxy);

/\* only one of the following activities can be called next, only once for the given ctx \*/

/\* the structure returned in \*new\_creds must be deallocated by the caller \*/

**CMP\_err** **CMPclient\_imprint**(**CMP\_CTX** \*ctx, **CREDENTIALS** \*\*new\_creds,

**const** **EVP\_PKEY** \*newkey, **const** **char** \*subject,

OPTIONAL **const** **X509\_EXTENSIONS** \*exts);

**CMP\_err** **CMPclient\_bootstrap**(**CMP\_CTX** \*ctx, **CREDENTIALS** \*\*new\_creds,

**const** **EVP\_PKEY** \*newkey, **const** **char** \*subject,

OPTIONAL **const** **X509\_EXTENSIONS** \*exts);

**CMP\_err** **CMPclient\_pkcs10**(**CMP\_CTX** \*ctx, **CREDENTIALS** \*\*new\_creds,

**const** **X509\_REQ** \*p10csr);

**CMP\_err** **CMPclient\_update**(**CMP\_CTX** \*ctx, **CREDENTIALS** \*\*new\_creds,

**const** **EVP\_PKEY** \*newkey);

/\* reason codes are defined in openssl/x509v3.h \*/

**CMP\_err** **CMPclient\_revoke**(**CMP\_CTX** \*ctx, **const** **X509** \*cert, **int** reason);

/\* must be called after any of the above activities \*/

**void** **CMPclient\_finish**(**CMP\_CTX** \*ctx);

/\* CREDENTIALS helpers \*/

**CREDENTIALS** \***CREDENTIALS\_new**(OPTIONAL **const** **EVP\_PKEY** \*pkey,

OPTIONAL **const** **X509** \*cert,

OPTIONAL **const** **STACK\_OF(X509)** \*chain,

OPTIONAL **const** char \*pwd,

OPTIONAL **const** char \*pwdref);

**void** **CREDENTIALS\_free**(OPTIONAL **CREDENTIALS** \*creds);

**X509** \***CREDENTIALS\_get\_cert**(**const** **CREDENTIALS** \*creds);  
**STACK\_OF(X509)** \***CREDENTIALS\_get\_chain**(**const** **CREDENTIALS** \*creds);

/\* certs is name of a file in PKCS#12 format; primary cert is of client \*/

/\* source for private key may be "[pass:<pwd>]" or "engine:<id>" \*/

**CREDENTIALS** \***CREDENTIALS\_load**(OPTIONAL **const** **char** \*certs,

OPTIONAL **const** **char** \*key,

OPTIONAL **const** **char** \*source,

OPTIONAL **const** **char** \*desc /\* for diagnostics \*/);

**bool** **CREDENTIALS\_save**(**const** **CREDENTIALS** \*creds,   
 OPTIONAL **const** **char** \*certs, OPTIONAL **const** **char** \*key,

OPTIONAL **const** **char** \*source, OPTIONAL **const** **char** \*desc);

**X509** \***CERT\_load**(**const** **char** \*file, OPTIONAL **const** **char** \*pass, OPTIONAL **const** **char** \*desc);

**bool** **CERT\_save**(**const** **X509** \*cert, **const** **char** \*file, OPTIONAL **const** **char** \*desc);

**X509\_REQ** \***CSR\_load**(**const** **char** \*file, OPTIONAL **const** **char** \*desc);

/\* LOG helpers \*/

**bool** **LOG**(OPTIONAL **const** **char** \*func, OPTIONAL **const** **char** \*file, **int** lineno,  
 **severity** level, **const** **char** \*fmt, ...);

**bool** **LOG\_default**(OPTIONAL **const** **char** \*func, OPTIONAL **const** **char** \*file, **int** lineno,

**severity level, const char \*msg);  
void** **LOG\_set\_verbosity**(**severity** level);

**void LOG\_set\_name**(OPTIONAL **const char** \*name);

**void** **LOG\_close**(**void**);

/\* X509\_STORE helpers \*/

**STACK\_OF(X509)** \***CERTS\_load**(**const** **char** \*files, OPTIONAL **const** **char** \*desc);

**int** **CERTS\_save**(**const** **STACK\_OF(X509)** \*certs, **const** **char** \*file, OPTIONAL **const** **char** \*desc);

**void** **CERTS\_free**(OPTIONAL **STACK\_OF(X509)** \*certs);

**STACK\_OF(X509\_CRL)** \***CRLs\_load**(**const** **char** \*file, **int** timeout, OPTIONAL **const** **char** \*desc);

**void** **CRLs\_free**(OPTIONAL **STACK\_OF(X509\_CRL)** \*crls);

**X509\_STORE** \***STORE\_load**(**const** **char** \*trusted\_certs, OPTIONAL **const** **char** \*desc);

**bool** **STORE\_add\_crls**(**X509\_STORE** \*truststore, OPTIONAL **const** **STACK\_OF(X509\_CRL)** \*crls);

/\* also sets certificate verification callback: \*/

**bool** **STORE\_set\_parameters**(**X509\_STORE** \*truststore,

OPTIONAL **const** **X509\_VERIFY\_PARAM** \*vpm,

**bool** full\_chain, **bool** try\_stapling,

OPTIONAL **const** **STACK\_OF(X509\_CRL)** \*crls,

**bool** use\_CDP, OPTIONAL **const** **char** \*cdps, **int** crls\_timeout,

**bool** use\_AIA, OPTIONAL **const** **char** \*ocsp, **int** ocsp\_timeout);

**typedef** **X509\_CRL** \*(\***CONN\_load\_crl\_cb\_t**)(OPTIONAL **void** \*arg,

OPTIONAL **const** **char** \*url, **int** timeout,

**const** **X509** \*cert, OPTIONAL **const** **char** \*desc);

**bool** **STORE\_set\_crl\_callback**(**X509\_STORE** \*store,

OPTIONAL **CONN\_load\_crl\_cb\_t** crl\_cb,

OPTIONAL **void** \*crl\_cb\_arg);

**bool STORE\_set1\_host\_ip**(**X509\_STORE** \*store, **const char** \*host, **const char** \*ip);

**void STORE\_free**(OPTIONAL **X509\_STORE** \*truststore);

/\* EVP\_PKEY helpers \*/

**EVP\_PKEY** \***KEY\_new**(**const** **char** \*spec); /\* spec may be "RSA:<length>" or "EC:<curve>" \*/

**EVP\_PKEY** \***KEY\_load**(OPTIONAL **const** **char** \*file, OPTIONAL **const** **char** \*pass,  
 OPTIONAL **const** **char** \*engine, OPTIONAL **const** **char** \*desc);

**void** **KEY\_free**(OPTIONAL **EVP\_PKEY** \*pkey);

/\* SSL\_CTX helpers for HTTPS \*/

**SSL\_CTX** \***TLS\_new**(OPTIONAL **X509\_STORE** \*truststore,  
 OPTIONAL **const** **STACK\_OF(X509)** \*untrusted,

OPTIONAL **const** **CREDENTIALS** \*creds,  
 OPTIONAL **const** char \*ciphers, **int** security\_level);

**void** **TLS\_free**(OPTIONAL **SSL\_CTX** \*tls);

/\* X509\_EXTENSIONS helpers \*/

**X509\_EXTENSIONS** \***EXTENSIONS\_new**(**void**);

/\* add optionally critical Subject Alternative Names (SAN) to exts \*/

**bool** **EXTENSIONS\_add\_SANs**(**X509\_EXTENSIONS** \*exts, **const** **char** \*spec);

/\* add extension such as (extended) key usages, basic constraints, policies \*/

**bool** **EXTENSIONS\_add\_ext**(**X509\_EXTENSIONS** \*exts, **const** **char** \*name,

**const** **char** \*spec,OPTIONAL **BIO** \*sections);

**void** **EXTENSIONS\_free**(OPTIONAL **X509\_EXTENSIONS** \*exts);

#endif /\* GENERIC\_CMP\_CLIENT\_H \*/