SIEMENS

Generic CMP client library API

– released, updated –

Version 1.2

Dr. David von Oheimb, Siemens CT RTA ITS SEA-DE

Restricted

Table of Contents

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

1. Introduction

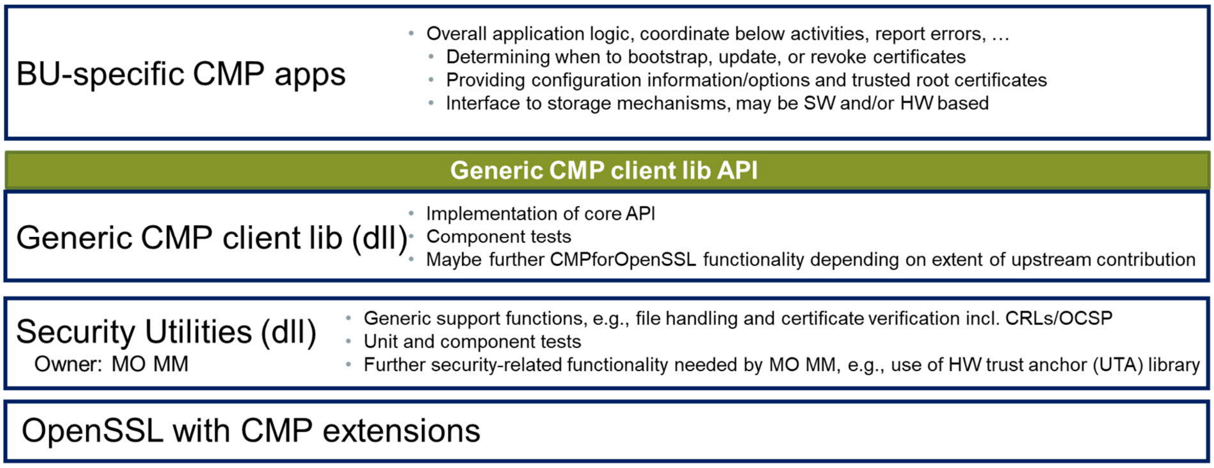
1.1 Motivation and Purpose

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 specifies the Application Programming Interface (API) for the development of a generic CMP client library to be used by Siemens business units (BUs) in the development of their products and solutions.

1.2 Scope

There is already 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 rather high-level command-line interface (CLI).


BU-specific CMP apps
Overall application logic, coordinate below activities, report errors, …
Determining when to bootstrap, update, or revoke certificates
Providing configuration information/options and trusted root certificates
Interface to storage mechanisms, may be SW and/or HW based
Generic CMP client lib API
Generic CMP client lib (dll)
Implementation of core API
Component tests
Maybe further CMPforOpenSSL functionality depending on extent of upstream contribution
Security Utilities (dll)
Owner: MO MM
Generic support functions, e.g., file handling and certificate verification incl. CRLs/OCSP
Unit and component tests
Further security-related functionality needed by MO MM, e.g., use of HW trust anchor (UTA) library
OpenSSL with CMP extensions

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. The library allows developing CMP clients that fulfill the Siemens Product PKI CMP profile [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 or as a standalone CMP library dynamically linked to OpenSSL (in all generally supported versions, currently 1.0.2., 1.1.0, and 1.1.1).

1.3 References

Reference ID	Document with title, unique identifier and version
[RFC4210]	RFC4210: “Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)“, 2005 https://www.ietf.org/RFC/RFC4210.txt
[RFC4211]	RFC4211: “Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)”, 2005 https://www.ietf.org/RFC/RFC4211.txt
[RFC5280]	RFC5280: “Certificate and Certificate Revocation List (CRL) Profile”, 2008 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
[cmpossl]	M. Peylo and D. von Oheimb: CMP for OpenSSL project https://github.com/mpeylo/cmpossl (code repository including documentation)
[CMP_Profile]	I. Wenda and D. von Oheimb: “CMP Profile for Siemens use cases”
[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
[CMP_Wiki]	Overview and general information on CMP on the CT PKI Product PKI Wiki https://wiki.ct.siemens.de/x/zYPdBw

2. Overview

This CMP client library is built on top of CMPforOpenSSL, which in turn is built on top of OpenSSL. 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.

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 name 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 frees 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(OPTIONAL log_fn);;
CMP_CTX *ctx = NULL;
…
err = CMPclient_prepare(&ctx, OPTIONAL log_fn,
                        OPTIONAL cmp_truststore, OPTIONAL untrusted,
                        OPTIONAL creds, OPTIONAL digest,
                        OPTIONAL transfer_fn, total_timeout,
                        OPTIONAL new_cert_truststore, implicit_confirm);
…
err = CMPclient_setup_HTTP(ctx, server, path, timeout, OPTIONAL tls, OPTIONAL proxy);
…
err = CMPclient_imprint(ctx, &new_creds, new_key, subject, OPTIONAL exts);
…
CMPclient_finish(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.

3. 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 actual 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 OSSL_cmp_log_cb_t log_fn);
```

3.1 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`	`OSSL_cmp_log_cb_t`	`log_fn`	Function to be called for logging CMP related errors, warnings, etc. See section 3.7 for details. If `NULL` is given `OSSL_CMP_puts()` is used, which prints errors and warnings to `stderr`, while info and debug messages are printed to `stdout`.
`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 STACK_OF(X509) *`	`untrusted`	Non-trusted intermediate CA certificates that may be needed for path construction during authentication of the CMP server and potentially others (i.e., TLS server and the newly enrolled certificate).
`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.

OPTIONAL	const char *	digest	Name of hash function to use when signing, for proof-of-possession (POPO) when requesting a certificate and also for protecting messages. The available digest names can be shown with the command openssl list -digest-commands
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.
	bool	implicit_confirm	Flag whether to request implicit confirmation

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

```
CMP_err err;
CMP_CTX *ctx = NULL;
OSSL_cmp_log_cb_t log_fn = NULL;
X509_STORE *cmp_truststore = …
STACK_OF(X509) *untrusted = NULL;
CREDENTIALS *creds = …
const char *digest = "sha256";
OSSL_cmp_transfer_cb_t transfer_fn = NULL; /* default HTTP(S) transfer */
int total_timeout = 100;
X509_STORE *new_cert_truststore = …
bool implicit_confirm = 0;
err = CMPclient_prepare(&ctx, log_fn,
                        OPTIONAL cmp_truststore, OPTIONAL untrusted,
                        OPTIONAL creds, OPTIONAL digest,
                        OPTIONAL transfer_fn, total_timeout,
                        OPTIONAL new_cert_truststore, implicit_confirm);
```

3.2 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 freed immediately. The optional tls parameter is not copied, so must not be freed before invoking CMPclient_finish().

Param.	Type	Name	Meaning
	CMP_CTX *	ctx	CMP context to be filled
	const char *	server	Server address, of the form "<name/ip>[:<port>]" with the default port being 8080
	const char *	path	Server HTTP path (aka CMP alias)
	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 proxy address, of the form "<name/ip>[:<port>]", with the default port being 8080. This argument may be overridden by the environment variable http_proxy. No proxy is used if the server name is found in the environment variable no_proxy. So far, HTTP proxies are not supported for TLS.

Example use (for the parts replaced by '…' and TLS_new(), see section 0):

```
const char *server = "ppki-playground.ct.siemens.com:443";
const char *path = "/ejbca/publicweb/cmp/PlaygroundECC ";
const char *proxy = NULL;
int timeout = 10;
X509_STORE *tls_truststore = …
CREDENTIALS *tls_creds = …
char *ciphers = TLS_get_ciphers(NULL); /* "HIGH:!ADH:!LOW:!EXP:!MD5:@STRENGTH" */
SSL_CTX *tls = TLS_new(tls_truststore, tls_creds, OPTIONAL ciphers, -1);
err = CMPclient_setup_HTTP(ctx, server, path, timeout, tls, OPTIONAL proxy);
```

3.3 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	Key (pair) to use for the new certificate; the private key is used for self-signature (POPO) and the corresponding public key is put in the cert 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 held in a crypto engine.
	const char *	subject	X.509 Subject Distinguished Name (DN) in the form "/<type0>=<value0>/<type1>=<value1>...".
OPTIONAL	const X509_EXTENSIONS *	exts	X.509 extensions to put in the certificate 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 *	csr	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, OPTIONAL 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, 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_encroll()`. For more flexibility these may be called also directly.

3.4 *CMPclient_update*

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

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

3.5 *CMPclient_revoke*

The function `CMPclient_revoke()` performs revocation (using the CMP command '`rr`') of the given certificate.

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`

3.6 *CMPclient_finish*

The function `CMPclient_finish()` deallocates the given CMP context, freeing all internal data but not the structures passed in via the functions described before. Due to current limitations of CMPforOpenSSL, 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 freed when not needed any more.

Param.	Type	Name	Meaning
	`CMP_CTX *`	`ctx`	CMP context to delete

Example use (for the various `_free()` functions, see sections 3.8 and 0):

```
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);
```

3.7 Logging callback function

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 a `file`, `lineno`, and `msg` argument that may provide a string and number indicating the source code location and gives 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 errors, warnings, debugging info, 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 int bool; /* false or true */
typedef enum {LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR,
              LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG} OSSL_CMP_severity;
typedef bool (*OSSL_cmp_log_cb_t) (OPTIONAL const char *file, int lineno,
                                   OSSL_CMP_severity level, const char *msg);
```

3.8 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 CMP_err (*OSSL_cmp_transfer_cb_t) (CMP_CTX *ctx, const OSSL_CMP_PKIMESSAGE *req,
                                           OSSL_CMP_PKIMESSAGE **res);
```

3.9 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 https://tools.ietf.org/html/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);
```

3.10 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 password-based authentication (such as PBM) the credentials include at least the password to use 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 two 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 free 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 PBMAC 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 frees the structure itself. It has no return value.

Param.	Type	Name	Meaning
OPTIONAL	`CREDENTIALS *`	creds	Credentials structure to delete

As long as we do not make the `CREDENTIALS` data type opaque there is no need to define selector functions; instead the components can be accessed directly, e.g., via `creds->pkey`.

4. 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.

4.1 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, could also be supported.

The function `CREDENTIALS_load()` reads from the file with the name given in the `certs` argument 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 with the name given in the `key` argument (which will typically be the same as `certs`), 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. 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 engine to use and the remaining credentials components are loaded from the file (without decrypting it). The respective engine must already have been parameterized and initialized in an engine-specific way with the usual OpenSSL mechanisms, which are described for instance in https://www.openssl.org/docs/man1.1.0/crypto/ENGINE_init.html

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(const char *certs, 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, OPTIONAL source,
                                      OPTIONAL 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,
                                          OPTIONAL tls_desc);
```

The function `CREDENTIALS_save()` writes the components of the given credentials data structure `creds` to the file given as the `file` argument. In case the `source` argument is `NULL` or begins with `“pass:”`, it stores the private key in the same file, where the `source` argument may refer to a password in the form `“pass:<pwd>”` that is then used to encrypt the credential contents including the private key before storing them in the given file. In case the `source` argument begins with `"engine:"`, it assumes that the private key is held in the given engine and there is no need (and also no possibility) for it to save the key nor to encrypt the remaining contents. The function returns `1` on success and `0` otherwise. In case of errors optionally the string held in the optional `desc` parameter is used for forming error messages.

```
bool CREDENTIALS_save(const CREDENTIALS *creds, const char *file,
                      OPTIONAL const char *source, OPTIONAL const char *desc);
```

An example use has already been given in section 3.3.

4.2 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 and initializes it 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. The function returns the pointer to the constructed trust store on success, or `NULL` otherwise. In case of errors optionally 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 `CRLs_load()` loads the CRL(s) held in the DER or PEM file(s) with the given comma-separated list of file names in the `files` argument 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.

```
STACK_OF(X509_CRL) *CRLs_load(const char *files, OPTIONAL const char *desc);
```

The function `STORE_set_parameters()` sets various optional verification parameters and callbacks in the given trust store `truststore`; in more detail:

- inherit any given OpenSSL certificate verification parameters `vpm`
- require certificate status checks for all certificates in a chain in case the `full_chain` option is set. Certificate status checks are required at least for the leaf (first) certificate in a chain in case any of the following options is set.
  For each certificate for which the status check is required the verification function will try to obtain the revocation status first from OCSP stapling if enabled, then from any locally available CRLs, then via OCSP if enabled, and finally via CDPs. Verification fails if no valid and current revocation status can be found or the status implies that the certificate has been revoked.
- enable OCSP stapling, which make sense only for TLS, if `try_stapling` is set
- add any CRLs provided in the `crls` argument and in this case enable CRL-based checks,

- enable CRL-based checks in case a static URL for fetching CRLs is given as the `CRLs_url` argument or the use of CDP entries in certificates is enabled via the `use_CDPs` argument (where `CRLs_url` is used as a fallback to these CDPs)
- enable the use of OCSP in case a static OCSP responder URL is given as the `OCSP_url` argument or the use of AIA entries in certificates is enabled via the `use_AIAs` argument (where `OCSP_url` is used as fallback to these AIAs).

The function returns `1` on success and `0` otherwise. Further non-default trust store parameters may be set as far as needed using the various respective low-level OpenSSL functions.

```
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_CDPs, OPTIONAL const char *CRLs_url,
                          bool use_AIAs, OPTIONAL const char *OCSP_url);
```

Example use 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, OPTIONAL desc);
bool use_CDPs = true;
const char *CRLs_url = NULL;
bool use_AIAs = false;
const char *OCSP_url = NULL;
bool success = STORE_set_parameters(truststore, OPTIONAL vpm,
                                    full_chain, try_stapling,
                                    OPTIONAL crls,
                                    use_CDPs, OPTIONAL CRLs_url,
                                    use_AIAs, OPTIONAL OCSP_url);
```

The function `STORE_free()` deletes the given trust store. It has no return value.

```
void STORE_free(OPTIONAL X509_STORE *truststore);
```

4.3 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>"`. where the RSA key length may be 1024, 2048, or 4096 and 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 held in a crypto engine need to be generated by other (engine-specific) means.

The function `KEY_free()` deletes the given key `pkey` and wipes its representation in memory if it is software-based. It has no return value. For HW-based keys it just deletes the reference.

```
void KEY_free(OPTIONAL EVP_PKEY *pkey);
```

4.4 SSL_CTX helpers

The function `TLS_new()` sets up a new OpenSSL `SSL_CTX` structure with reasonable default parameters for HTTPS client connections. See also the Siemens TLS configuration [TLS_Conf] and integration guidelines [TLS_Integration]. Its optional arguments are the trust

store ts to use for authenticating TLS servers, the credentials creds to use for the client to authenticate to TLS servers, and the allowed cipher suites. All these parameters are not consumed, so should be freed by the caller. Yet the trust store should not be freed until the TLS connection has been closed (and is CMPclient_finish() called) because otherwise diagnostic information in case of host name mismatch would not be available.
The available cipher suite names can be shown with the command openssl list -cipher-algorithms. See also https://www.openssl.org/docs/man1.1.0/apps/ciphers.html 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 https://www.openssl.org/docs/man1.1.0/ssl/SSL_CTX_get_security_level.html.
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 const X509_STORE *truststore,
                 OPTIONAL const CREDENTIALS *creds,
                 OPTIONAL const char *ciphers, int security_level);
```

The function TLS_free() deletes the given TLS context tls. It has no return value.

```
void TLS_free(OPTIONAL SSL_CTX *tls);
```

4.5 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 1 on success and 0 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 https://www.openssl.org/docs/manmaster/man5/x509v3_config.html), which can be provided via the optional sections parameter. The function returns 1 on success and 0 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 http://oidref.com/1.3.6.1.5.5.7.3.

```
bool EXTENSIONS_add_ext(X509_EXTENSIONS *exts, const char *name,
                        const char *spec, BIO *sections);
```

Possibly further such functions will be added later.

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()` deletes the given structure `exts`. It has no return value.

```
void EXTENSIONS_free(OPTIONAL X509_EXTENSIONS *exts);
```

5. Appendix: C header file

```
/*!****************************************************************************
 * @file   genericCMPclient.h
 * @brief  generic CMP client library API
 *
 * @author David von Oheimb, CT RDA ITS SEA, David.von.Oheimb@siemens.com
 *
 * @copyright (c) Siemens AG, 2018. The Siemens Inner Source License - 1.1
 ******************************************************************************/

#ifndef GENERIC_CMP_CLIENT_H
#define GENERIC_CMP_CLIENT_H

/* for low-level CMP API, in particular, type CMP_CTX */
#include <openssl/cmp.h>
typedef OSSL_CMP_CTX CMP_CTX; /* for abbreviation and backward compatibility */

/* error codes are defined in openssl/cmperr.h */
typedef int CMP_err; /* should better be defined and used in openssl/cmp.h */
#define CMP_OK 0

#define CMP_IR    OSSL_CMP_PKIBODY_IR
#define CMP_CR    OSSL_CMP_PKIBODY_CR
#define CMP_P10CR OSSL_CMP_PKIBODY_P10CR
#define CMP_KUR   OSSL_CMP_PKIBODY_KUR
#define CMP_RR    OSSL_CMP_PKIBODY_RR

#ifndef __cplusplus
typedef enum { false = 0, true = 1 } bool; /* Boolean value */
#endif

#define OPTIONAL /* this marker will get ignored by compiler */

/* private key and related certificate, plus optional chain */
typedef struct credentials {
    EVP_PKEY *pkey;                    /* can refer to HW key store via engine */
    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;

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);


/* CMP client core functions */
/* should be called once, as soon as the application starts */
CMP_err CMPclient_init(OPTIONAL OSSL_cmp_log_cb_t log_fn);

/* must be called first */
CMP_err CMPclient_prepare(CMP_CTX **pctx,
                          OPTIONAL OSSL_cmp_log_cb_t log_fn,
                          OPTIONAL X509_STORE *cmp_truststore,
                          OPTIONAL const STACK_OF(X509) *untrusted,
                          OPTIONAL const CREDENTIALS *creds,
                          OPTIONAL const char *digest,
                          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 */
/* copies server and proxy address (of the form "<name>[:<port>]") and HTTP path */
CMP_err CMPclient_setup_HTTP(CMP_CTX *ctx, const char *server, const char *path,
                             int timeout, OPTIONAL SSL_CTX *tls,
                             OPTIONAL const char *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 freed 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 *csr);
CMP_err CMPclient_update(CMP_CTX *ctx, CREDENTIALS **new_creds,
                         const EVP_PKEY *newkey, const X509 *old_cert);
/* 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 */
/* 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(const char *certs, const char *key,
                              OPTIONAL const char *source,
                              OPTIONAL const char *desc/* for error msgs */);
/* file is name of file to write in PKCS#12 format */
bool CREDENTIALS_save(const CREDENTIALS *creds, const char *file,
                      OPTIONAL const char *source, OPTIONAL const char *desc);


/* X509_STORE helpers */
/* trusted_certs is name of a file in PEM or PKCS#12 format */
X509_STORE *STORE_load(const char *trusted_certs,
                       OPTIONAL const char *desc/* for error msgs */);
STACK_OF(X509_CRL) *CRLs_load(const char *file, OPTIONAL const char *desc);
/* 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_CDPs, OPTIONAL const char *CRLs_url,
                          bool use_AIAs, OPTIONAL const char *OCSP_url);
void STORE_free(OPTIONAL X509_STORE *truststore);


/* EVP_PKEY helpers */
/* spec may be "RSA:<length>" or "EC:<curve>" */
EVP_PKEY *KEY_new(const char *spec);
void KEY_free(OPTIONAL EVP_PKEY *pkey);


/* SSL_CTX helpers for HTTPS */
SSL_CTX *TLS_new(OPTIONAL const X509_STORE *truststore,
                 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 */
```