



EUROPEAN MIDDLEWARE INITIATIVE

COMMON AUTHENTICATION LIBRARY – DEVELOPER'S GUIDE

Document version:	**2.1.2**
EMI Component Version:	**2.x**
Date:	**August 10, 2013**

This work is co-funded by the European Commission as part of the EMI project under Grant Agreement INFSO-RI-261611.

Copyright © EMI. 2010-2013.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CONTENTS

1 INTRODUCTION **4**
1.1 LANGUAGE BINDINGS 4
1.2 GETTING AND BUILDING LIBRARY 4
1.3 GENERAL GUIDELINES 4
1.4 CONTEXT AND PARAMETER SETTINGS 5

2 CANL COMPONENTS **5**
2.1 HEADER FILES 5
2.2 BUILDING CLIENT PROGRAMS 5
2.3 CONTEXT 5

3 CLIENT-SERVER AUTHENTICATED CONNECTION **7**
3.1 MAIN API WITHOUT DIRECT CALLS TO OPENSSL 7
3.2 MAIN API WITH DIRECT CALLS TO OPENSSL 9
3.3 SECURE CLIENT-SERVER CONNECTION EXAMPLE 9

4 CREDENTIALS HANDLING **11**
4.1 CERTIFICATE API 11
4.2 MAKE NEW PROXY CERTIFICATE – EXAMPLE 14

1 INTRODUCTION

This document serves as a developer's guide and could be seen as an API reference too, even though comments in the header files may give the reader better insights into that matter.

Common Authentication Library (caNl for short) was designed to provide common security layer support in grid applications. It is largely based on existing code (VOMS, LB). Its simple API can be devided by functionality into two parts:

- *caNl Main API* is used to establish (secure) client-server connection with one or both sides authenticated, send or receive data. As will be described in 4, most of the *Main API* is not directly dependent on some chosen cryptography toolkit (SSL implementation). It is also internally plugin-based and therefore other security mechanisms support can be added in future.
- *caNl Certificate API* allows certificate and proxy management for example proxy creation, signing, etc. We may think of *Certificate API* as the second level of *Main API*

Currently there is EMI Product Team assigned to caNl development with three subgroups for each language binding.

1.1 LANGUAGE BINDINGS

caNl is developed in C language as well as C++ and Java language bindings, however this document covers only the C interface.

1.2 GETTING AND BUILDING LIBRARY

TODO package names

external dependencies:

- c-ares – asynchronous resolver library
- openssl – cryptography and SSL/TLS toolkit

1.3 GENERAL GUIDELINES

Naming conventions

All function names are prefixed with `canl_`

Input and output arguments

All structures and objects passed in output of functions (even though pointers are used as a help) are dynamically allocated, so proper functions to free the allocated memory has to be called. e.g. `canl_free_ctx()` deallocates members of the structure `canl_ctx`.

Opaque types

Almost all types used in caNl are *Opaque types* – i.e. their structure is not exposed to users. To use and/or modify these structures API call has to be used. Example of opaque type is `canl_ctx`.

Return values

The return type of most of the API functions is `canl_err_code` which in most cases can be interpreted as int. Unless specified otherwise, zero return value means success, non-zero failure. Standard error codes from `errno.h` are used as much as possible.

Few API functions return `char *`. In such a case `NULL` indicates an error, non-null value means success.

1.4 CONTEXT AND PARAMETER SETTINGS

All the API functions use a *context* parameter of type `canl_ctx` to maintain state information like error message and code. Some API functions also use an *io context* of type `canl_io_handler` which keeps information about each particular connection (for example socket number, oid, SSL context).The caller can create as many contexts as needed, all of them will be independent. When calling `canl_create_ctx()` or `canl_create_io_handler()` all members of the objects are initialized with default values which are often NULL for pointer type and 0 in case of int and similar types.

2 caNl COMPONENTS

2.1 HEADER FILES

Header files for the common structures and functions are summarized in table 1.

`canl.h`	Definition of context objects and *Main API* common functions declarations.
`canl_ssl.h`	Declaration of functions that use X509 certificates based authentication mechanism (pretty much dependent on openssl library functions).
`canl_cred.h`	Definition of context objects of the *Certificate API* and functions declarations.

Table 1: Header files

2.2 BUILDING CLIENT PROGRAMS

The easiest way to build programs using caNl in C is to use GNU's libtool to take care of all the dependencies:

```
libtool --mode=compile gcc -c example1.c -D_GNU_SOURCE
libtool --mode=link gcc -o example1 example1.o -lcanl_c
```

2.3 CONTEXT

Context initialization

There are two opaque data structures representing caNl *Main API* context: `canl_ctx` and `canl_io_handler` (see section 1.4). `canl_ctx` must be initialized before any caNl API call. `canl_io_handler` must be initialized before calling function representing io operation (for example `canl_io_connect()`) and after `canl_ctx` initialization.

```
#include <canl.h>
#include <canl_ssl.h>

canl_io_handler my_io_h = NULL;
canl_ctx my_ctx;
my_ctx = canl_create_ctx();
err = canl_create_io_handler(my_ctx, &my_io_h);
```

There is one opaque data structure representing caNl *Certificate API* context: `canl_cred`. It must only be initialized before function calls that use this context as a parameter.

```
#include <canl.h>
#include <canl_cred.h>

canl_ctx ctx;
canl_cred c_cred;
ctx = canl_create_ctx();
canl_cred_new(ctx, &c_cred);
```

Obtaining error description

`canl_ctx` stores details of all errors which has occurred since context initialization, in human readable format. To obtain it use `canl_get_error_message()`:

```
printf("%s\n", canl_get_error_message(my_ctx));
```

Context deallocation

It is recommended to free the memory allocated to each context if they are not needed anymore, in first case `canl_io_handler` , then `canl_ctx` in case of the *Main API*:

```
if (my_io_h)
        canl_io_destroy(my_ctx, my_io_h);
canl_free_ctx(my_ctx);
```

as for the Certificate API:

```
canl_cred_free(ctx, c_cred);
```

3 CLIENT-SERVER AUTHENTICATED CONNECTION

For client-server authenticated connection we just use caNl *Main API* calls. In time of writing this paper caNl use *openssl – SSL/TLS and cryptography toolkit*. However, core of the caNl has been developed to be as independent on any cryptography toolkit as possible, so it may support other libraries in the future.

3.1 MAIN API WITHOUT DIRECT CALLS TO OPENSSL

These are the functions of the *Main API* that do not use *openssl API* calls or variable types directly (as a parameter or in their definitions):

- `canl_ctx canl_create_ctx()`

  This function returns an initialized *authentication context* object

- `void canl_free_ctx(canl_ctx cc)`

  This function will free the *authentication context*, releasing all associated information. The context must not be used after this call.

  – param cc – the *authentication context* to free

- *canl error code* `canl_create_io_handler(canl_ctx cc, canl_io_handler *io)`

  This function will create an *i/o handler* from the *authentication context*. This handler shall be passed to all I/O-related functions.

  – param cc – the *authentication context*
  – param io – return an initialized *i/o context*, or NULL if it did not succeed
  – return – caNl error code

- `canl_err_code canl_io_close(canl_ctx cc, canl_io_handler io)`

  This function will close an existing connection. The 'io' object may be reused by another connection. It is safe to call this function on an io object which was connected.

  – param cc – the *authentication context*
  – param io – the *i/o context*
  – return – *canl error code*

- `canl_err_code canl_io_connect(canl_ctx cc, canl_io_handler io, const char *host, const char *service, int port, gss_OID_set auth_mechs, int flags, struct timeval *timeout)`

  This function will try to connect to a server object, doing authentication (if not forbidden)

  – param cc – the *authentication context*
  – param io – the *i/o context*
  – param host – the server to which to connect
  – param service – the service on the server - usually NULL
  – param port – the port on which the server is listening

  - param auth_mechs – authentication mechanism to use
  - param flags – for future usage
  - param peer – if not NULL the canl_principal will be filled with peer's principal info. Approppriate free funcion should be called if canl_princ is no longer to be used
  - param timeout – the timeout after which to drop the connect attempt
  - return – *canl error code*

- `canl_err_code canl_io_accept(canl_ctx cc, canl_io_handler io,int fd, struct sockaddr s_addr, int flags,canl_principal *peer, struct timeval *timeout)`

  This function will setup a server to accept connections from clients, doing authentication (if not forbidden)

  - param cc – the *authentication context*
  - param io – the *i/o context*
  - param fd – file descriptor to use
  - param port – the port on which the server is listening
  - param sockaddr – open socket address
  - param flags – for future usage
  - param peer – if not NULL the canl_principal will be filled with peer's principal info. Approppriate free funcion should be called if canl_princ is no longer to be used
  - return – *canl error code*

- `canl_err_code canl_princ_name(canl_ctx cc, const canl_principal cp, char **ret_name)`

  Get the peer's principal name in text readable form.

  - param cc – the *authentication context*
  - param cp – canl structure to hold peer's principal info. Have to be filled by previous call to canl_io_accept or canl_io_connect funcions.
  - param ret_name – text form of the peer's princ. name
  - return – *canl error code*

- `void canl_princ_free(canl_ctx cc, canl_principal cp)`

  If canl_princ structure filled before by some canl io funcion, this function should be called to free the allocated memory.

  - param cc – the *authentication context*
  - param cp – canl peer's principal structure
  - return – void

3.2 MAIN API WITH DIRECT CALLS TO OPENSSL

- `canl_err_code canl_ctx_set_ssl_cred(canl_ctx cc, char *cert, char *key, char *proxy, canl_password_callback clb, void *pass)`

  This function will set the credential to be associated to the *context*. These credentials will become the default ones for all API calls depending on this *context*.

  - param cc – the *authentication context*
  - param cert – the certificate to be set
  - param key – its private key
  - param proxy – the proxy certificate to be set
  - param clb – a callback function which should return the password to the private key, if needed
  - param pass – user specified data that will be passed as is to the callback function. Note that the content of this pointer will not be copied internally, and will be passed directly to the callback. This means that altering the data pointed by it will have a direct effect on the behavior of the function.
  - return – *canl error code*

- `canl_err_code canl_ctx_set_ca_dir(canl_ctx cc, const char *ca_dir)`

  Set certficate authority directory (openssl ca directory structure)

  - param cc – rhe *authentication context*
  - param ca_dir – the path that will be set. It will not be checked whether this path actually contains the CAs or not
  - return – *canl error code*

- `canl_err_code canl_ctx_set_crl_dir(canl_ctx cc, const char *crl_dir)`

  - param cc – the *authentication context*
  - param crl_dir – the path that will be set. It will not be checked whether this path actually contains the CRLs or not
  - return – *canl error code*

- `canl_err_code canl_ctx_set_ssl_flags(canl_ctx cc, unsigned int flags)`

  Set SSL specific flags. This function can turn OCSP check ON. (OFF by default)

  - param cc – the *authentication context*
  - param flags – one of the canl_ctx_ssl_flags in canl_ssl.h (e.g. CANL_SSL_OCSP_VERIFY_ALL)
  - return – *canl error code*

3.3 SECURE CLIENT-SERVER CONNECTION EXAMPLE

We give an example of a caNl client that use *Main API* with openssl. We do not define variables in this example, unless their type is caNl defined. For complete sample see `canl_samples_server.c` in source package or canl_sample_server.c at CVS

Include nesessary header files:

```
#include <canl.h>
#include <canl_ssl.h>
```

Initialize context and set parameters:

```
canl_ctx my_ctx;
canl_io_handler my_io_h = NULL;
my_ctx = canl_create_ctx();
err = canl_create_io_handler(my_ctx, &my_io_h);
err = canl_ctx_set_ssl_cred(my_ctx, serv_cert, serv_key, NULL, NULL);
```

set user's credentials (X509 auth. mechanism)

```
if (serv_cert || serv_key || proxy_cert){
        err = canl_ctx_set_ssl_cred(my_ctx, serv_cert, serv_key, proxy_cert,
                                    NULL, NULL);
        if (err) {
            printf("[CLIENT] cannot set certificate or key to context: %s\n",
                    canl_get_error_message(my_ctx));
            goto end;
        }
}
```

If using X509 auth. mechanism, we might set *CA directory* and/or *CRL directory* at this place. (If not set, default directories will be used, that is those in proper env. variables) . . .

Connect to the server, send something then read the response:

```
err = canl_io_connect(my_ctx, my_io_h, p_server, NULL, port, NULL, 0, &timeout);
if (err) {
        printf("[CLIENT] connection to %s cannot be established: %s\n",
               p_server, canl_get_error_message(my_ctx));
        goto end;
    }
err = canl_io_write (my_ctx, my_io_h, buf, buf_len, &timeout);
if (err <= 0) {
        printf("can't write using ssl: %s\n",
               canl_get_error_message(my_ctx));
        goto end;
    }
err = canl_io_read (my_ctx, my_io_h, buf, sizeof(buf)-1, &timeout);
if (err > 0) {
        buf[err] = '\0';
        printf ("[CLIENT] received: %s\n", buf);
        err = 0;
    }
```

Free the allocated memory:

```
if (my_io_h)
        canl_io_destroy(my_ctx, my_io_h);
canl_free_ctx(my_ctx);
```

4 CREDENTIALS HANDLING

If we want to create new proxy certificate or for example delegate credentials, we can use caNl *Certificate API*. This part of API uses X509 authentication mechanism (openssl library now)

4.1 CERTIFICATE API

These are the functions of the *Certificate API*, all of them use `canl_ctx` as first parameter and `canl_err_code` as a return value, so we do not include them in following description:

- `canl_err_code canl_cred_new(canl_ctx, canl_cred *cred)`

  This function creates new structure (context) to hold credentials.

  - param cred – a new object will be returned to this pointer after success

- `canl_err_code canl_cred_free(canl_ctx, canl_cred *cred)`

  This function will free the credentials context, releasing all associated information. The context must not be used after this call.

  - param cred – the credentials context to free

- `canl_err_code canl_ctx_set_cred(canl_ctx, canl_cred cred)`

  This one sets users credentials to caNl context.

  - param cred – credentials to set to global caNl context

- `canl_err_code canl_cred_load_priv_key_file(canl_ctx, canl_cred cred, const char * file, canl_password_callback clb, void *pass)`

  Load private key from specified file into the credentials context.

  - param cred – credentials which save private key to
  - param file – the file to load private key from
  - param clb – the callback function which should return the password to the private key, if needed.
  - param pass – User specified data that will be passed as is to the callback function

- `canl_cred_load_chain(canl_ctx, canl_cred cred, STACK_OF(X509) *chain)` This function loads the certificate chain out of an openssl structure. The chain usually consist of a proxy certificate and certificates forming a chain of trust.

  - param cred – the credentials context to set chain to
  - param chain – the openssl structure to load certificate chain from.

- `canl_cred_load_chain_file(canl_ctx, canl_cred cred, const char * file)` This function loads the certificate chain out of a file. The chain usually consists of a proxy certificate and certificates forming a chain of trust.

  - param cred – credentials which save certificate chain to
  - param file – the file to load certificate chain from

- `canl_err_code canl_cred_load_cert(canl_ctx, canl_cred cred, X509 *cert)`

  This function loads user certificate out of an openssl structure

  - param cred – the credentials context to set certificate to
  - param cert – the openssl structure to load certificate from

- `canl_err_code canl_cred_load_cert_file(canl_ctx, canl_cred cred, const char *file)`

  This function loads user certificate out of a file.

  - param cred – credentials which save certificate to
  - param file – the file to load certificate from

- `canl_err_code canl_cred_set_lifetime(canl_ctx, canl_cred cred, const long lt)`

  This function sets the lifetime for a certificate which is going to be created

  - param cred – the credentials context
  - param lt – the lifetime in seconds

- `canl_err_code canl_cred_set_extension(canl_ctx, canl_cred cred, X509_EXTENSION *ext)`

  This function sets the certificate extension to for the certificate which is going to be created

  - param cred – the credentials context
  - param ext – the openssl structure holding X509 certificate extension

- `canl_err_code canl_cred_set_cert_type(canl_ctx, canl_cred cred, const enum canl_cert_type type)`

  This function sets the certificate type to for the certificate which is going to be created.

  - param cred – the credentials context
  - param type – a canl_cert_type in canl_cred.h

- `canl_err_code canl_cred_sign_proxy(canl_ctx, canl_cred signer, canl_cred proxy)`

  This function makes new proxy certificate based on information in *proxy* parameter. The new certificate is signed with private key saved in *signer*. A new certificate chain is saved into *proxy*.

  - param signer – the credentials context which holds signer's certificate and key.
  - param proxy – the credentials context with a certificate signing request, public key and user certificate; optionally lifetime, certificate type and extensions.

- `canl_err_code canl_cred_save_proxyfile(canl_ctx, canl_cred cred, const char * file)`

  This function saves proxy certificate into a file.

  - param cred – the credentials context with certificate to save
  - param file – save the certificate into

- `canl_err_code canl_cred_save_cert(canl_ctx, canl_cred cred, X509 **to)`

  This function saves certificate into openssl object of type *X509*

  - param cred – the credentials context with certificate to save
  - param to – save the certificate into

- `canl_err_code canl_cred_save_chain(canl_ctx, canl_cred cred, STACK_OF(X509) **to)`

  This function saves certificate chain of trust with proxy certificate into openssl object of type *STACK_OF(X509)*.

  - param cred – the credentials context with certificate chain to save
  - param to – save the certificate into

- `canl_err_code canl_cred_new_req(canl_ctx, canl_cred cred, unsigned int bits)`

  This function creates a new certificate signing request after a new key pair is generated.

  - param cred – the credentials context, certificate signing request is saved there
  - param bits – the key length

- `canl_err_code canl_cred_save_req(canl_ctx, canl_cred cred, X509_REQ **to)`

  This function saves certificate signing request into openssl object of type *X509_REQ*.

  - param cred – the credentials context with certificate request
  - param to – save the certificate request into

- `canl_err_code canl_cred_save_req(canl_ctx, canl_cred cred, X509_REQ **to)`

  This function loads certificate signing request from openssl object of type *X509_REQ* into caNl certificate context

  - param cred – the credentials context, the cert. request will be stored there
  - param to – load the certificate request from

- `canl_err_code canl_verify_chain(canl_ctx ctx, X509 *ucert, STACK_OF(X509) *cert_chain, cha`

  Verify the certificate chain, openssl verification, CRL, OCSP, signing policies etc...

  - param ucert – user certificate
  - param cert_chain – certificate chain to verify
  - param cadir – CA certificate directory

- `canl_err_code canl_verify_chain_wo_ossl(canl_ctx ctx, char *cadir, X509_STORE_CTX *store_c`

  Verify certificate chain, SKIP openssl verif. part; Check CRL, OCSP (if on), signing policies etc. (This is special case usage of caNl, not recommended to use unless you really know what you are doing)

  - param cadir – CA certificate directory
  - param store_ctx – openssl store context structure fed with certificates to verify

4.2 MAKE NEW PROXY CERTIFICATE – EXAMPLE

We give an example of a proxy certificate creation. We do not define variables in this example, unless their type is caNl defined. We do not check return values in most cases as well. For complete sample see example sources.

Include necessary header files:

```
#include <canl.h>
#include <canl_cred.h>
```

caNl context variables

```
canl_cred signer = NULL;
canl_cred proxy = NULL;
canl_ctx ctx = NULL;
```

Initialize context:

```
ctx = canl_create_ctx();
ret = canl_cred_new(ctx, &proxy);
```

Create a certificate request with a new key-pair.

```
ret = canl_cred_new_req(ctx, proxy, bits);
```

(Optional) Set cert. creation parameters

```
ret = canl_cred_set_lifetime(ctx, proxy, lifetime);
ret = canl_cred_set_cert_type(ctx, proxy, CANL_RFC);
```

Load the signing credentials

```
ret = canl_cred_new(ctx, &signer);
ret = canl_cred_load_cert_file(ctx, signer, user_cert);
ret = canl_cred_load_priv_key_file(ctx, signer, user_key, NULL, NULL);
```

Create the new proxy certificate

```
ret = canl_cred_sign_proxy(ctx, signer, proxy);
```

And store it in a file

```
ret = canl_cred_save_proxyfile(ctx, proxy, output);

if (signer)
    canl_cred_free(ctx, signer);
if (proxy)
    canl_cred_free(ctx, proxy);
if (ctx)
    canl_free_ctx(ctx);
```