The OpenCA manual

..., Michael Bell <loon@openca.org>

January 19, 2001

Contents

Part I

Introduction

Have somebody the time to write a small essay?

Part II

OpenCA::DBI and DBIS

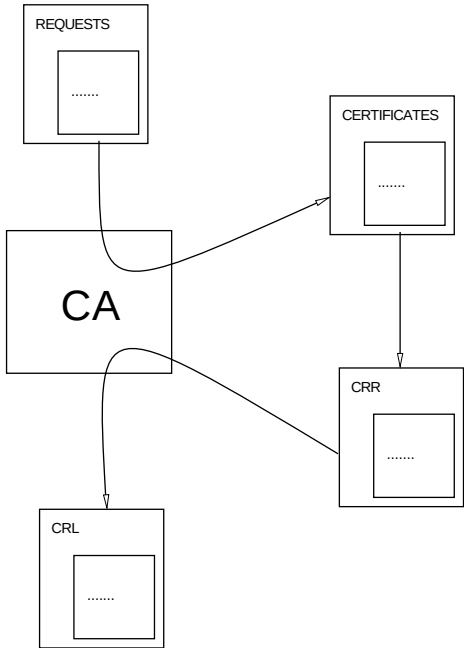
Chapter 1

Design of OpenCA::DBI and DBIS

1.1 Databasedesign

1.1.1 Lifecycle like used for planning the modules

Before we start to design the databases it is a good idea to make an image which shows the lifecycle of a certificate.



1.1.2 Lifecycle analysis

The lifecycle show us that there are four major objects in a certificate's lifecycle:

1. Request
2. Certificate
3. CRR
4. CRL

Not every certificate must be revoked and so the most certificates will never be part of a CRR or a CRL but we should take about every possibility. So in the database we have to store a minimum of four instances of objects.

1.1.3 Logging

Systems which should provide security must have very improved logging to support a very deterministic behaviour. If we have any problems like deleted requests, revoked certificates which are never published on a CRL we want to follow the trace of actions.

1.1.4 Security

In this case security is the security of the logs. Requests, certificates, CRRs and CRLs protect themselves by signing. The logs are not protected so timestamps can be manipulated some logs can be removed. there are a lot of things which can happen. So I have to sign the logs that nobody can manipulate old entries. If anyone cracks my database and remove all the stuff before time 1234 then I can do nothing only regular backups help.

1.1.5 Databasedesign

The basic database consists of six tables.

1. request
2. certificate
3. crr
4. crl
5. log
6. signature

I never use any plurals. Plurals can only be misinterpreted. So I take the basic type to give the tables their names and this is the object's name. The structures of the tables are the followings:

request

name	datatype
REQUEST_SERIAL	TEXT PRIMARYKEY
ROLE	TEXT
DATA	TEXT
FORMAT	VARCHAR32
INFO	TEXT
DN	TEXT
CN	TEXT
EMAIL	TEXT
RA	TEXT
RAO	TEXT
STATUS	SMALLINT (a bug?)

certificate

name	datatype
CERTIFICATE_SERIAL	BIGINT PRIMARYKEY
ROLE	TEXT
FORMAT	VARCHAR32
DATA	TEXT
INFO	TEXT
DN	TEXT
CN	TEXT
EMAIL	TEXT
STATUS	SMALLINT (a bug?)

ca_certificate

name	datatype
CA_CERTIFICATE_SERIAL	TEXT PRIMARYKEY
ROLE	TEXT
FORMAT	VARCHAR32
DATA	TEXT
INFO	TEXT
DN	TEXT
CN	TEXT
EMAIL	TEXT
STATUS	SMALLINT (a bug?)

crr

name	datatype
CRR_SERIAL	TEXT PRIMARYKEY
ROLE	TEXT
CERTIFICATE_SERIAL	BIGINT
DATE	DATETIME
FORMAT	VARCHAR32
DATA	TEXT
INFO	TEXT
DN	TEXT
CN	TEXT
EMAIL	TEXT
RA	TEXT
RAO	TEXT
STATUS	SMALLINT (a bug?)
REASON	TEXT

crl

name	datatype
CRL_SERIAL	DATETIME PRIMARYKEY
ROLE	TEXT
STATUS	SMALLINT (a bug?)
FORMAT	VARCHAR32
DATA	TEXT
LAST_UPDATE	DATETIME
NEXT_UPDATE	DATETIME
CRL_SERIAL	SERIAL
DATATYPE	TEXT (evtl. redundant with FORMAT)
INFO	TEXT

log

name	datatype
ACTION_NUMBER	SEQUENCE
MODULETYPE	SMALLINT
MODULE	TEXT
SUBMIT_DATE	DATETIME
DO_DATE	DATETIME
ACTION	BIGINT
CERTIFICATE_SERIAL	BIGINT
REQUEST_SERIAL, CRR_SERIAL CA_CERTIFICATE_SERIAL	TEXT
CRR_SERIAL, DATE	DATETIME
ROLE	ROLE
FORMAT	VARCHAR32
DATA	TEXT
INFO	TEXT
DN	TEXT
CN	TEXT
EMAIL	TEXT
RA	TEXT
RAO	TEXT
LAST_UPDATE	DATETIME
NEXT_UPDATE	DATETIME
DATATYPE	TEXT (evtl. redundant with FORMAT)
STATUS	SMALLINT (a bug?)
REASON	TEXT

signature

name	datatype
ACTION_NUMBER	BIGINT PRIMARYKEY
CERTIFICATE_SERIAL	BIGINT
DATE	DATETIME
DATA	TEXT
INFO	TEXT

Attention - if you are looking into your database you will not find these columnnames directly because in the module itself every name has a extra databasename. This was done to provide a fast change of the hole database if there are problems with a special database. So for example I use *key* as name for the serial of requests, CA-certificates and CRRs but MySQL doesn't like the word *key* as variablename so I change the name to *mykey*.

Another nasty detail is actually that I unify the unique identificatiers. This means all unique integer identifiers (CRLs and certificates) are named *serial* and all hashes are named *key* (ok now *mykey ;-)*).

The datatypes are not real datatypes the show you the meaning and we have special settings for every different databasevendor. For example: PostgreSQL can have sometimes problems with ISO-date so the datatype DATETIME is in real *text* for PostgreSQL.

1.1.6 Plans related to the database for the future

The most important projects in the future are OpenCA::SYNC and the design of a RBAC-concept. The concept is explained in the RBAC-section too (Part IV).

1.2 OpenCA::DBI and OpenCA::DBIS

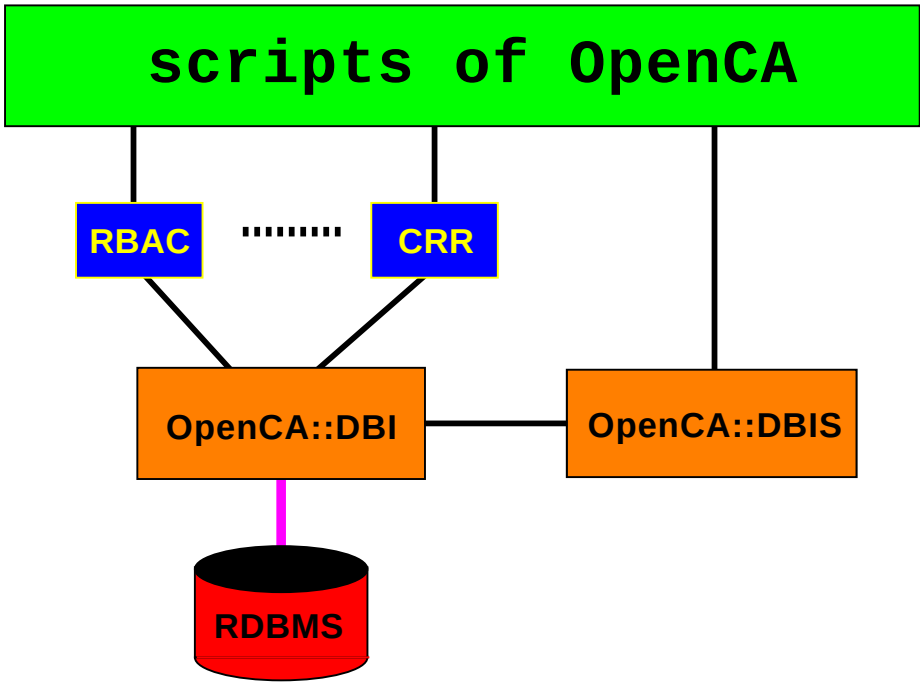
1.2.1 History

Originally OpenCA::DBI was developed to support the OpenCA project with an interface to RDBMSs which should be compatible with OpenCA::DB. During the development the need for logsigning will be integrated into *OpenCA::DBI* and I made the decision to build a second module *DBI Services.*

The OpenCA::DBIS software includes all highlevel functionality which are needed to use a RDBMS. This includes backup, logsigning, datamerging before signing, calculating the "anchors" for a logentry and all of the verification mechanisms. Actually only signing is implemented. There are no concepts today to implement backup and verification. These are with the RBAC- and SYNC-modules the most wanted features which we need to reach the level of commercial systems.

1.2.2 Basic designideas

First a small overview how the design was planned. This show the way of the objects! So the scripts access of cause the OpenCA::DBI module but the passed data is for example a CRR-object. Another thing is that some other modules like RBAC uses functions of OpenCA::DBIS too (e.g. the function getMergedData). The RBAC for example can use the signing daemon on the CA to give the admins at the CA a little bit more comfort. They have only to enter the passphrase one times (at start of the CA)



1.2.3 Which code in which module?

The basic idea is to put all functions which operate directly on the database to the OpenCA::DBI module and all functions which implements special features are put to OpenCA::DBIS. therefore the *S* means *Services*. So if you want to use a highlevelfunction like the daemon which handle the signing you have to look into the interfacedocumentation of OpenCA::DBIS.

1.2.4 Open for the future

The modules especially the DBI-module are so open as possible. Which means they mainly object oriented - not in the meaning of programming but in the meaning of the processed data. So you give the module an object and the module stores the object in the correct databasetable. So it should be no problem to integrate the new RBAC-objects. The entries which are needed for databasesynchronisation are today there (it's only the *DO_DATE* - the signature of the log will be automatically overwritten).

Chapter 2

Internal documentation

2.1 OpenCA::DBI

2.1.1 usage

- $new_object = new OpenCA::DBI (option1 => $value1, ...);
- All names are case sensitive !!!

2.1.2 used packages

- OpenCA::REQ
- OpenCA::X509
- OpenCA::CRL
- OpenCA::CRR
- OpenCA::OpenSSL
- OpenCA::Tools
- DBI
- OpenCA::DBIS

2.1.3 Configuration and creation of object by new ()

This is perhaps the most complicated part for the users of this module. You configure and init a new object of the class OpenCA::DBI by calling the function new.

The often used remote and local means remote database and local database. Actually OpenCA does not have a sync-module so the use of a local database insteed of the central remote database makes absolut no sense. The code for

the documented options is written but deactivated by enforced special settings of the options.

ALL VARIABLES ARE CASESENSITIVE !!!

The usage of sub new is:

SHELL => $object object of class OpenCA::OpenSSL

mode => mode_name

The following modes are available

mode	write	access	read	access
	standard	backup	standard	backup
ultra-secure	remote	-	remote	-
secure	remote	-	remote	local
standard	remote	-	local	-
progressive	remote	local	local	-
agressive	local	-	local	-

Because the synchronization doing module of OpenCA is not implemented so "ultra-secure" is enforced. Attention the code is implemented for this feature so if take the code and remove the line

$self->{mode} = "ultra-secure";

,the described mechanisms work.

failsafe => (on|off)

If failsafe is *on* and a remote action fails a failover by using the local database is encouraged. If mode is *"ultra-secure"* this option will be ignored.

Actually failsafe is ever set to "off". (Code is written and works - so you can activate this feature.)

second_chance => (yes|no)

If no backup is set due to the settings of mode and failsafe a *"yes"* enforce a second try on the standarddatabase. A failed read during the first run don't effect a failing write during the second run so a third run will be performed.

Actually *"no"* is enforced but manually you can made this option working. The enforced *"no"* is only there for better testing (definite setting at every time).

logsecurity => integer_value

The default setting is 0. You can set integer value higher than -1 (so >=0).

logperformance => integer_value

The default setting is 8. You can set integer value higher than 7 (so >=8).

So this is the time to write something about the signing code. Actually the code is placed in OpenCA::DBI but in future it moves to OpenCA::DBIS which stands for DBI Services - so it includes all high level functionality.

The log signing algorithm create signature from the following logrecords:

-> actual

-> actual - 2**0

-> actual - 2**1

-> ...

-> actual - 2**k (last value with result >= 0)

1. Now the records 0..(logperformance-1) are ignored.
2. The records logperformance..t are removed until k-t equals logsecurity. If logsecurity equals 0 then the second operation is not performed and all records are signed.

logperformance is an option which protects us against the possibilities of systems which has a high system load factor. logsecurity defines the used references.

remoteType => Pg

DBI type of the database

remoteName => database_name

Be warned this string is a must for every databasesystem! Several databases does not need the setting of host or port because the information is stored in an interfaces file like on Sybase. So the *"database_name"* is the string needed by the databasedrivers of the different vendors. For more information please the the documentation of the DBD::vendor_name drivers (e.g. Informix, Interbase, mSQL, MySQL, Oracle, Pg, Sybase)

remoteHost => hostname

This is the host where the database is located - so remote has only a logical meaning. Actually until you use no VPN-software it is strongly recommended that the database is on your local machine. The use of DNS is not necessary and not recommended because the use of pure IP protects you against DNS spoofing. Alternatively you can insert the used hostname in your /etc/hosts

remotePort => port_number

remoteUser => user

remotePasswd => passwd

localType => Pg

localName => database_name

localHost => hostname

localPort => port_number

localUser => user

localPasswd => passwd

DEBUG => true_value

If you enter nothing then there is nothing. If you enter a value which perl interprets as true then debugging is on.

CERT_FILE => cert.pem This is for logging only. The cert is used for signing. Please see CONFIGURATION - new () for more information about the algorithm.

KEY_FILE => priv.key This is for logging only. The key is used for signing. Please see CONFIGURATION - new () for more information about the algorithm.

PASSWD => passwd This is for logging only. The passphrase is used for signing. Please see CONFIGURATION - new () for more information about the algorithm.

Last a small comment to the code which do the signing. Because this code is a highlevel feature I will move this in the future to the OpenCA::DBIS module so not be shocked if you are looking into the code and don't find the code. If the code moved you will find a notice about the version were the move starts here.

Version: 0.1.4.14

THE VALUES OF MODE, FAILSAFE AND SECOND_CHANCE ARE NOT CASE SENSITIVE.

2.1.4 Datastructures and variables

variable	content
$OpenCA::DBI::SQL	all sql stuff
$OpenCA::DBI::SQL->{TABLE}	all tables
$OpenCA::DBI::SQL->{VARIABLE}	all variables [0] (as array with type [1])
$OpenCA::DBI::SQL->{TABLE_STRUCTURE}	array with columnnames of tables
$OpenCA::DBI::ACTION	actioncodes for logging
$OpenCA::DBI::STATUS	status of a object
$OpenCA::DBI::MODULETYPES	ra, ca, public ...
$OpenCA::DBI::ERROR	all errorcodes
$OpenCA::DBI::DB	contains all vendorspecific configuration
$OpenCA::DBI::DB->{Pg}	PostgreSQL
$OpenCA::DBI::DB->{Pg}->{TYPE}	types
$OpenCA::DBI::DB->{Pg}->{DBI_OPTION}	options for DBI->connect
$OpenCA::DBI::DB->{Pg}->{SEQUENCE}	stuff for sequence generators
$OpenCA::DBI::VERSION	versionnumber

The variables which are bind to the object itself are all named like above described in 2.1.3.

2.1.5 Public functions

The supported public functions are:

new - see CONFIGURATION -new ()

Please see the description of the configuration of OpenCA::DBI which describe the "new" function.

initDB

This function initializes the databases. It knows the following options:

DB => @databases You can pass an array which can include "remote", "local" or "remote" and "local". If nothing is included then the value is set to "remote". The databases will then initialized.

This means the function tries to do all the sql-create commands which are needed for operation of the OpenCA::DBI and OpenCA::DBIS modules.

These tables are:

request certificate crr crl log signature

MODE => (NONE|FORCE|FORCE_LOCAL|FORCE_REMOTE|FORCE_ALL)

If successful then the function returns a 0. If not successfull then -1 is returned. Please read this section carefully because I perhaps switch to returnvalue 1 for success. Comments are welcome.

operateTable

DO=> (exist|drop|creat|init)

DB => $dsn this is *$self->{remoteDB}* or *$self->{localDB}*

TYPE => $db_type

TABLE => $table Sequencegenerators are handled extra!

storeItem

DATATYPE => (old_type|basic_type) The old_types which are accepted are the same like in the OpenCA::DB module. These are strings like PENDING_REQUEST or REVOKED_CERTIFICATE.

The basic_type means you can enter normal basic types like:

- REQUEST
- CERTIFICATE
- CRR
- CRL

If you use basic types and you not set the option "status" status is setting to "VALID". If you use old_types then the status will be extracted from the string via the private function getStatus.

STATUS => (VALID|RENEWED|UPDATED|PENDING|APPROVED|SUSPENDED|REVOKED|DELETED|ARCHIVED|EXPIRED|) The status can be any of the above terms. If status is not seeded I use first the DATATYPE if it is an old_type and if not not then the status is "VALID".

INFORM => (PEM|DER|SPKAC|) This option is actually a little bit unclear because I get the data via objects so I don't need the format because I get the data directly from the object. If the format is not detectable I have an internal variable defStoredFormat which define this format.

Résumé: this is waste!

OBJECT => $openca_object This is an OpenCA object which has to be stored. This could be OpenCA::REQ OpenCA::X509 OpenCA::CRR OpenCA::CRL

MODULETYPE => (CA|PKIManager|RA|WebGateway|) This for logging only. If you set it you can read the log in the database and can verify via OpenCA::DBI::MODULETYPE->{number_from_db} the moduletype which has done this action.

MODULE => module_name This is for logging only. If you set it you can read the log in the database and can verify which module has done this action (it is stored as ascii so it is humanreadable - means you can read it as databaseadmin).

getItem

DATATYPE => (old_type|basic_type) The old_types which are accepted are the same like in the OpenCA::DB module. These are strings like PENDING_REQUEST or REVOKED_CERTIFICATE.

The basic_type means you can enter normal basic types like:

- REQUEST
- CERTIFICATE
- CRR
- CRL

If you use basic types and you not set the option "status" status is setting to "VALID". If you use old_types then the status will be extracted from the string via the private function getStatus.

STATUS => (VALID|RENEWED|UPDATED|PENDING|APPROVED|SUSPENDED|REVOKED|DELETED|ARCHIVED|EXPIRED|) The status can be any of the above terms. If status is not seeded I use first the DATATYPE if it is an old_type and if not then the status is ignored.

KEY => key This is the key (the unique identifier) of this special requested object. So this can be a serial number or a md5 etc..

If KEY is not given then I return the last element. This feature is useful for CRLs and only actually allowed for CRLs!!! If you search the latest one you have only to call:

$openca_dbi->getItem (DATATYPE => "CRL");

I think this is a good feature.

If you need this feature for other objects you must uncomment the following line in getItem:

return if ((not $serial) && ($table ne "CRL"));

MODE => (RAW|) RAW causes the return of the plain text of stored data. Nothing causes the return of an object.

getNextItem

The same options like getItem except MODE which is not supported. An object will be returned at every time. The function determines only the next key itself and then passes the request to the function getItem. The option KEY is required.

getPrevItem

The same options like getItem except MODE which is not supported. An object will be returned at every time. The function determines only the next key itself and then passes the request to the function getItem. The option KEY is required.

destroyItem

DATATYPE => (old_style|basic_type)

KEY => key destroyItem really delete the request from the database. Attention this function is reserved for a fututre recovery algorithm! therefore the operation will not be logged!

So please "hands off" if you not very shure what you are doing!!!

Use deleteItem (which do nothing ;-)) or better (best)

storeItem (DATATYPE= xyz, MODE=>"UPDATE", STATUS=>"DELETED", OBJECT=>xyz);

deleteItem

This is a dummy to be proof against old codeparts which think they must remove the object from VALID_CERTIFICATE after they store the certificate to REVOKED_CERTIFICATE.

elements

DATATYPE => (old_type|basic_type) The old_types which are accepted are the same like in the OpenCA::DB module. These are strings like PENDING_REQUEST or REVOKED_CERTIFICATE.

The basic_type means you can enter normal basic types like:

- REQUEST
- CERTIFICATE
- CRR
- CRL

If you use basic types and you not set the option "status" the function returns the number of all elements of this table.

STATUS => (VALID|RENEWED|UPDATED|PENDING|APPROVED|SUSPENDED|REVOKED|DELETED|ARCHIVED|EXPIRED|) If not used the scan performs on the hole table.

This function counts the elements which are in the same table and have the same status (if status is set via STATUS or DATATYPE).

searchItem

The options are the well known options DATATYPE, MODE and STATUS (please see above).

The new options are all possible searchattributes. To get them please use the new function getAttributes! The old functions support some types not. The function getAttribute don't return the unique identifiers, but you can get the unique identifiers of the tables via OpenCA::DBI::SQL- >{VARIABLE}->{tablename."_SERIAL"}[0] (Attention - the tablename is stored in big letters!)

getTimeString

This function returns an ISO-timestring (2001-01-14 18:24:06).

2.1.6 Unchanged public functions (from OpenCA::DB v0.8.7a)

rows

Same options like searchItem. The function calls searchItem and count the returned objects. Simple but errorproof

Working but unclear status (private or public???) (directly taken from OpenCA::DB v0.8.7a)

listItem

This function is directly taken over from OpenCA::DB v0.8.7a. Because I don't know for what it is used I don't change and use it.

The following unsupported functions are not supported because they perform operations which are not necessary or possible for RDBMSs (Relational DataBase Management Systems). These systems take care by themselves on things like number of elements, locks, next and preview operators etc..

2.1.7 Private functions

The "new" private functions are:

getSearchAttributes

The only argument is the tablename via getAttributes ("REQUEST"); The returned value is an array with the available attributes. Take in mind that the unique identifiers will not be returned but they are available via OpenCA::DBI::SQL->{VARIABLE}->{CERTIFICATE_SERIAL}[0] for example.

getTable

It extract from a datatype (old or new) the tableand return it.

getStatus

It extracts from STATUS and DATATYPE the status. If STATUS is present DATATYPE will be ignored.

getSequence

This function has the job to return a new ACTION_NUMBER for the table log. This is done by a function to keep the vendordependent code away from the not vendordependent code. Sequences, sequence generators etc. are not standardized. The option is a db_hash_write called hash. Pleae see doConnect for a detailed description of this code.

doConnect, doQuery, doRollback, doCommit, doDisconnect

All of these function get an hash as option. The hash is structured like follows:

```
my %db_hash_read = (STATUS => 0,
                    ERRORS => 0,
                    DBH => 0,
                    STH => [],
                    TYPE => "",
                    MODE => "READ",
                    QUERY => "",
                    BIND_VALUES => [] );
```

STATUS is the errorstate of the read and write connections. Please never touch this value it is absolut internal highly critical.

ERRORS include all errors which are happened during the use of this hash. If you want to use write and read connections you can merge the errors between the hashes via $db_hash_read{ERRORS} |= $other{ERRORS}; If you now return with return $db_hash_read{ERRORS} because doConnect has failed finally all database realted errors are detectable. All errors are available via $openca_dbi_object::ERROR- >{error_name}. Actually I think I have a problem because the |= operator does not the same thing like in C or I have some overflows.

DBH is the actual used databasehandle from DBI->connect.

STH is an array with all statementhandles of the actual DBH. The handle for the last doQuery is available via $hash{STH}[scalar (@{$hash{STH}}) -1]. Actually I don't use other than the last result of a statement but somewhere in the future ...

TYPE the DBD:driver - e.g. Pg, Informix, Oracle

MODE set the mode of the databaseoperation and so I can determine the used database which is defined via the function new (MODE=>"ultrasecure" ...)

QUERY this is the actual query which you have only to set for doQuery.

BIND_VALUES this is the actual array of binded values which you have only to set for doQuery.

2.1.8 Has to be implemented (!!!)

These functions are not absolut necessary because Massimiliano Pala explains me that der is uuencoded and PEM is nothing else then DER in txt-format.

txt2der

der2txt

The last two functions require a bidirectional equivalent transformation from binary data into text. This is necessary to store DER-formated data. I try something but it is not correct.

2.1.9 Old unchanged private functions (from OpenCA::DB v0.8.7a):

getBaseType

listItems (not used)

byKey (not used)

hash2txt

txt2hash

2.1.10 Supported databases

Every subscribed item has the same behaviour for remoteXYZ and localXYZ.

PostgreSQL

option	default	required
remoteType	Pg	yes
remoteName	-	yes
remoteHost	localhost	no
remotePort	5432	no
remoteUser	-	yes
remotePasswd	-	yes

If you would not set the remoteUser then DBD::Pg would use the username of the processowner. Because this is special for the Pg-driver this feature is not supported or used by the OpenCA::DBI-module and cause an undef return value for the new () call.

MySQL

Attention the name which you must enter is mysql!!!

option	default	required
remoteType	mysql	yes
remoteName	-	yes
remoteHost	localhost	no
remotePort	?	no
remoteUser	-	yes
remotePasswd	-	yes

Because I have not the time to test MySQL please write any mistake in this documentation suddenly to me. I don't know the standard MySQL-Port so I hope the DBD::mysql module knows it ;-)

ATTENTION - MYSQL AND MSQL DON'T SUPPORT TRANSACTIONS, SO PLEASE USE THEM ONLY FOR TESTING. THE OPENC::DBI MODULE WORKS ONLY CORRECT WITH RDBMSS WHICH FULLY SUPPORT TRANSACTIONS. THE OPENCA::DB-MODULE IS PREPARED TO HANDLE THE PROBLEM OF NONEXISTING ACID-FEATURES BECAUSE IT USES DBM-FILES. IF YOU WANT TO USE AN OPENSOURCE-DATABASE WITH OPENCA PLEASE TAKE POSTGRESQL OR ANY OTHER TRANSACTIONSUPPORTING RDBMSS. (WHAT IS WITH INTERBASE - ISN'T THIS THE FREE BORLAND^{TM} DB ?)

2.1.11 How to add another databasevendor?

The variable OpenCA::DBI::DB

First you have to edit the section where *$OpenCA::DBI::DB* is defined. Make a copy from the *$OpenCA::DBI::DB=>{Pg}* part and then start edit it.

TYPE The first problem is the date. I try to store a ISO-date so you have to decide whether your database can handle ISO-format or you choose better a type like "text".

DBI_OPTION this is the string which I give to DBI->connect

SEQUNCE This is a little bit difficult. This part is used to support sequence-generators. So there are two different ways.

1. If the sequencegenerator is implemented via auto_increment or a row where you can directly enter a value from a sequencegenerator like in PostgreSQL this is the right way to read.
   - CREATE - here you have to enter the sql-string which can create the the generator if this is explicitly necessary like in PostgreSQL and not in MySQL.
   - INIT - some generators need an initial command for startup.

- GENERATE - here you enter the code with which you fill the row of the generator (see the examples of PostrgeSQL and MySQL)
- GENERATE_BY_INSERT = 1 - in case 1.
- DROP - the sql-command to delete the generator if this is explicitly necessary (e.g. Oracle)

2. If the sequence generator must be eplicitly asked for a value then this is the correct way.
   - CREATE - see 1.
   - INIT - see 1.
   - GENERATE - nothing
   - GENERATE_BY_INSERT = 0
   - DROP - see 1.

The function getSequence

1. (cont.) here you must enter the code to get the value from the row which you have inserted with the GENERATE command. The value must be returned via return *$number;*

2. (cont.) here you have to enter the code to get the next number from your sequencegenerator.

The sub new ()

Please look for the comment

After this comment you find a section where the dsn of DBI (attention not OpenCA::DBI) for the DBD-driver will be configured. This is very easy. So have a look to PostgreSQL and MySQL and do it then.

You don't want to do this job?

No problem. Mail a message to me with a short comment which database do you need. Then I try to make it available.

MySQL was implemented one day after it was requested - so sometimes we are very fast ;-D

Michael Bell <michael.bell@web.de>

2.1.12 How to add another attribute?

1. Check in $OpenCA::DBI::SQL->{VARIABLE} for the existence of such an attribute. If it is still missing then add it.

2. Add the attribute to the table in the array $OpenCA::DBI::SQL->{TABLE_STRUCTURE} but take in mind that the first entry is the primary key.

3. Add the attribute to the table "LOG" in the array $OpenCA::DBI::SQL->{TABLE_STRUCTURE}->{LOG} but take in mind that the first entry is the primary key and especially here reserved for the sequencegenerator.

4. Add the attriubte to the list in getSearchAttributes.

So now the attriubte should be added. I think that's easy enough now ;-D

2.1.13 Authors of the software

- Massimiliano Pala <madwolf@openca.org> (c) 1997-2000, All Rights reserved.

- Michael Bell <michael.bell@web.de> (c) 2000-2001, All Rights reserved.

2.1.14 LICENSE

GNU Public License version 2. The parts are used from Massimiliano Pala's OpenCA::DB have a special license so please see OpenCA::DB for more information.

2.1.15 P.S. EXAMPLE

```
Block: {

         doConnect

         if doConnect returns negative

         then last BLOCK (final error, all options failsafe,

                  backup or second_chance did not help - best

                  thing is now to say return -1; insteed of

                  last BLOCK;)

          doQuery until the first returncode is -1

          then doRollback doDisconnect

          if never doQery fails then doCommit

          if returnvalue is -1

          then doRollback doDisconnect else doDisconnect

          if somethig fails except doConnect "next BLOCK"

         }
```

you can repeat this block so often as you want until the first time doConnect returns <0. So long this not happens you can try to get a successful transaction.

2.2 OpenCA::DBIS

2.2.1 Used packages

- OpenCA::REQ
- OpenCA::X509
- OpenCA::CRL
- OpenCA::CRR
- OpenCA::OpenSSL

- OpenCA::Tools
- DBI
- English (for nice names like $UID and $GID)
- POSIX (the setuid command is used but senseless because it only works for Perl v5.61 or higher)
- IPC::SysV
- IPC::SysV qw (IPC_RMID IPC_CREAT)

2.2.2 Datastructures and variables

Before I start with the table I want to notice here that the GLOBAL variables store mostly default values. The values which are accessible via $self->... are the actual values.

Global variables

variable	content
$OpenCA::DBIS::VERSION	versionnumber
$OpenCA::DBIS::ERROR	errorcodes
$OpenCA::DBIS::MESSAGEKEY	the key for the messagequeue (default: 673622324)
$OpenCA::DBIS::MESSAGELENGTH	the messagelength (unsused actual) (default: 256)
$OpenCA::DBIS::LOGSECURITY	0 (OpenCA::DBI for documentation)
$OpenCA::DBIS::LOGPERFORMANCE	8 (OpenCA::DBI for documentation)

Object's variables

variable	content
backend	undef
CERT_FILE	undef
KEY_FILE	undef
PASSWD	undef
MESSAGEKEY	$OpenCA::DBIS::MESSAGEKEY
MESSAGELENGTH	$OpenCA::DBIS::MESSAGELENGTH
PIDFILE	"/var/run/openca_signing_daemon.pid"
LOGFILE	"/var/log/openca_signing_daemon.log"
IPC_USER	undef
IPC_GROUP	undef
IPC_UID	undef
IPC_GID	undef
tools	undef (unused)
DEBUG	0

2.2.3 Public functions

new

calls $self->init

startSigningDaemon

1. calls $self->init
2. try to set uid and gid to the given IPC_UID and IPC_GID which are from IPC_USER and IPC_GROUP (this code works actually not on many machines because of a heavy bug in the POSIX package of perl which is only fixed for perl >=v5.61)
3. check for certificate and private key
4. try to get messagequeue
5. fork-parent
   (a) check access and existence of logfile
   (b) write pidfile
6. fork-child (endless loop)
   (a) get message from queue
   (b) fork-parent

i. begin the loop again

(c) fork-child

i. read from tmpfifoIn given by message all needed data
ii. call $self->getData
iii. call $self->getSigndata
iv. send message via answer_fifo taken from message

The daemon can handle errors during fork which means the signature is created by the parentprocess itself. The daemon logs every such problem because this can only happen if the computer has a too high load. The throughput is slowing down if this takes in place because the fifo limits the throughput which is a I/O-bottleneck!

stopSigningDaemon

This function calls only $self->init and then kills the daemon and removes the mesaagequeue. The messagequeue is removed to prevent a new daemon from trying to answer old request which are no longer existent. This would cause several hanging processes. So perhaps we have to change startSigningDaemon in the way to use IPC_EXCL or IPC_CREAT | IPC_TRUNC to completely avoid such problems.

getSignature

This is the function which contact the daemon. The function ignores any perhaps existent object and wants to get all option by iself. therefore it doesn't use sub init. You can send the following arguments:

- CERT_FILE
- KEY_FILE
- PASSWD
- SHELL
- MESSAGEKEY
- MESSAGELENGTH
- DATA
- DEBUG

If you send *CERT_FILE* and *KEY_FILE* the function doesn't use the daemon. The function use the daemon only if no certificate or private key are given. The functions implements the complete functionlaity for IPC by itself (means the function doesn't use their private functions for this).

getMergedData

merge the data from a transmitted hashreference (for use with the statementhandles - *$STH->fetchrow_hashref*). The function exists to provide all other functions which need to store a signature with an algorithm tho merge data in a secure manner to verify later the stored signatures.

getSignatureAnchor

calculate from a given position and given logsecurity and logperformance (described in *OpenCA::DBI*) all other entries in the table which should be used to build a secure signature and protect the database from removing some entries in the middle of the tables (actually only used to protect the table *log*).

2.2.4 Private functions

getSigndata

sign the transmitted data with transmitted certificate and privatekey

init

reads and interprets all arguments

- MESSAGEKEY
- MESSAGELENGTH
- CERT_FILE
- KEY_FILE
- PASSWD
- SHELL
- IPC_USER
- IPC_GROUP
- PIDFILE
- LOGFILE

doLog

write the transmitted message to the logfile with date. In the case of problems with the logfile the message is written to STDOUT.

debug

print the transmitted message and the complete configuration of the object.

getData

read the answer_fifo and the data which should be signed from the given fifo

2.2.5 dbis

This is the script which should be placed in the runlevel sections (/etc/init.d, rc? ...). You can configure all important variables of the signing daemon in the script. The script exports the variable and call dbisctl for startup. Used language is bash.

Because actually the setuid-command of the POSIX-class from Perl doesn't work (only >=5.61) I recommend to use this script not during the bootup (sudo - $IPC_USER is not implemented yet. Perhaps I do this later to fix the problems with perl. It is a bad style to store a passphrase in a startupscript. I think we better do startup via a cgi-script which uses dbisctl or include it.)

2.2.6 dbisctl

The script is written in Perl and uses the given variables in the environment. The script is thought for use with dbis.

Chapter 3

Configuration

3.1 DBI.conf

This file configures OpenCA::DBI and OpenCA::DBIS.

messagekey this is the key which is used to get the id of the messagequeue

messagelength unused (perhaps later limits the length of messages)

cert_file from OpenCA::DBIS

key_file from OpenCA::DBIS

passwd from OpenCA::DBIS

remoteType

remoteName

remoteHost

remotePort

remoteUser

remotePasswd

localType

localName

localHost

localPort

localUser

localPasswd

DEBUG

pidfile

logfile

The user and group are not specified because DBI.conf is only for use with cgi-scripts and the user and group of the webserver are the exactly right ones for the signing daemon.

ATTENTION PLEASE DON'T USE THE VARIABLES WHICH BEGIN WITH LOCAL BECAUSE THESE ARE ONLY USABLE IF OPENCA::SYNC WAS IMPLEMENTED. THIS IS ACTUALLY NOT HAPPEN.

Part III

OpenCA::Sync

somewhere in a far future ...

Part IV

OpenCA::ACL, ROLE and RBAC (not implemented yet!!!)

Attention - completely outdated !!!

Chapter 4

Design

4.1 Basic ideas

4.1.1 What we need?

The most important problem what we actually have is the accesscontrol. We only can devide into CA, RA, RA-Admin and Others (meaning: the public webserver). So we can only devide into functional modules but anyone who can use the module can use the hole functionality of the module (because today the RA-Admin and RA are implemented in the same script every RAO who knows the internal commands of OpenCA can use the functionality of the RA-Admin). So we need a strong mechanism to manage the access to the system and validate operations. On big systems I want only to say to the CA "every webserverrequest which was approved by the webadminstrators can be processed to a certificate". This is done by all good commercial systems via RBAC-models. There are several different ways to implement and interpret such models but what we need?

So to answer this question I have to summarize shortly what is needed to perform an operation:

1. an object like OpenCA::xyz
2. the status of the object
3. the role which will be affected
4. the operation which will be performed
5. the role which try to perform this operation

So the next question is how to determine a role? The role of an object is very easy to determine because any object in the system relate to a DN and every DN is like the name implies distinguished. The only extra role is a crl because the owner of a CRL is the CA so we have to declare a special role CA but this

is not a problem. The problem is to implement in every object the information about the role into the header.

For systems which has to guarantee nearly absolute security (ok this is never possible) we have to know only positive accessrights which means I don't have to store deny-rules only allow-rules exist - what is not allowed that is forbidden.

4.1.2 Designideas

So the main question is how looks a query which awaits as answer *deny* or *allow*?

So the input data is:

- datatype
- key
- dn of performer
- operation

What do we know from the input data?

datatype

- type of object
- status of object

key

- role of the object via getItem with dataype too

dn of performer

- certificate's validity
- role of performer

operation

- operation

Conclusion

So this looks like a very easy structure and yes it is not so difficult. I think we need basically three things:

1. an extension of our certificates to cleanly include the role (please not over the ou!)

2. a place for all roles
3. a place for the ACL(s)

So there are two possible ways to implement the roles and ACLs.

1. store all roles in a table to verify the existence of the role and store the ACL of a role in a seperate table for each role
2. store all roles and ACLs in one table

The decision which variant I want to take is not made because of performance. The existence of a table per role requires the right to create tables on the database. The operations are normally performed via a webpage with a cgi-script on the backend. Some databases allow the command *"create table"* only for superusers but I think it's not so good to hardcode a superuserpassphrase.

If you cannot accept that you as an owner of a highend database like DB2 or Oracle have to take care about problems with the accesscontrol of smaller databases be cool the second variant is faster because I can determine the right by one SQL-query to one table. So the possibility to have this table in the databasecache is very high (especially for databases with good cachealgorithms because this special table is accessed every time if a operation should be performed).

A small comment to the signature of a role. Every role and every accessright must be signed by the CA indiviually which means that the CA must sign every row of the table. It is theoretically possible to perform the signing about the hole ACL but then I need very long to verify a right because I must get all rights from the database.

If you find here any failure please contact the OpenCA group because this is very critical for the security of the hole system.

4.2 Databasetables

There is only one additional table:

name	datatype
RBAC_SERIAL	BIGINT
ROLE	TEXT
RIGHT	TEXT
OBJECT	TEXT
STATUS	SMALLINT (a bug?)
OWNER (affected role)	TEXT
OPERATION (future status)	SMALLINT
FORMAT	VARCHAR32
DATA (signature)	TEXT
INFO (only for compatibility)	TEXT

ATTENTION - there should be a row for every role inside the table where *RIGHT*, *OBJECT*, *OWNER* and *OPERATION* are empty (STATUS includes the status of the role - never delete a role completely via *OpenCA::DBI->destroyItem* to protect the integrity of the log!!!). We should store there a signature about the hole role to detect attacks where a right will be removed but not by the CA and there we can check the existence of a role which has no right (e.g. *User*). The signature of such a role is only performed about the single entry of this role. (I think it is not problematically if a hacker removes a right. The system can only be more secure in this way or the attack is very early recognized so actually I sign only the row itself to provide better performance and easier implementation.)

4.3 Functionality

Chapter 5

Internal documentation

5.1 Used packages

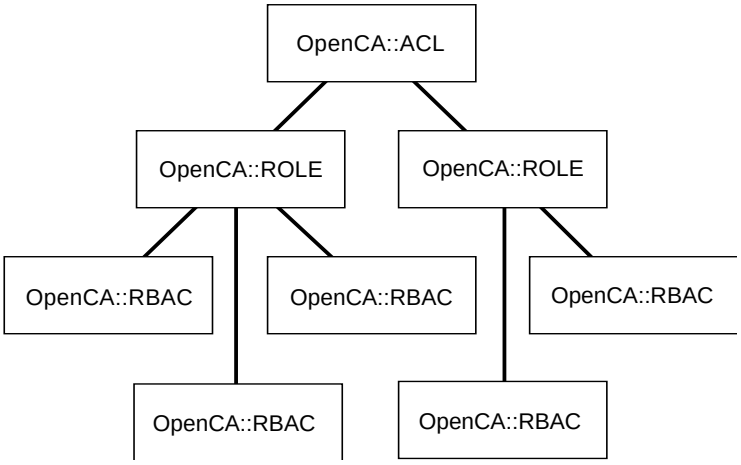
- OpenCA::DBI
- OpenCA::DBIS
- OpenCA::OpenSSL
- OpenCA::DBI and DBIS are a problem

5.2 Datastructures and variables

This is the most important section of the internal documentation. There are three big points which we have to support:

1. OpenCA::RBAC must be an object which I can handle with OpenCA::DBI
2. OpenCA::ROLE must be able to load a complete role
3. OpenCA::RBAC, ROLE and ACL must include the functionality to import and export the hole RBAC-model

So we must support the follwing structure:



So I can handle any of the needed three types of *OpenCA::RBAC*.

STATUS and OPERATION are both from the same source $OpenCA::DBI::STATUS{CODEWORD|N
So the combination of STATUS and OPERATION give us the ordinary operation (e.g. VALID (STATUS) to REVOKED (OPERATION) on a CERTIFICATE (OBJECT) of RAO (OWNER) is the revocation of RAO-certificate). The allowed status are:

- VALID
- RENEWED
- UPDATED
- PENDING
- APPROVED
- SUSPENDED
- REVOKED
- DELETED
- ARCHIVED
- EXPIRED
- EXIST

5.3 ACL

5.4 ROLE

5.5 RBAC

5.5.1 Public functions

new

All not commented options are taken directly from the databasetable

import

export

sign

done via getMergedData and getSignature from OpenCA::DBIS

create

initializes a new object of the given type (by mode)

store

changes all transmitted value

delete

delete the object from the database

load

removed because this functionality is done via getItem ;-D

getParsed

returns $self->{RIGHT. The parsing itself is done by *init*

getAccess

This function is the normally used one to test the existence of a right. You can enter:

- dn (transformed to role)
- type of object (used for table)
- serial/key of the object

1. I check the existence and state of the certificate matching the dn via getItem.
2. I do a getItem to check for the existence or status of the object.
3. I do a call searchItem via db-interface to find a matching right(attention: wildcardattacks –> before this operation you must verify the certificate of the user!).

5.6 Private functions

init

ITEM => $txtitem

This is only for the OpenCA::DB or OpenCA::DBI modules. This forces OpenCA::RBAC to ignore all other options go into mode RIGHT and start interpreting the TX-TITEM

RBAC_SERIAL

ROLE

If you set this variable and you set mode to ROLE then all roleobjects are loaded. If you don't set DB then this is an error an returns undefined!

RIGHT

OBJECT

STATUS

OWNER

OPERATION

FORMAT

DATA

INFO

DB => $databaseobject

You can choose between OpenCA::DB and OpenCA::DBI.

DAEMON => (ON|OFF|YES|NO|)

default is OFF, not necessary - unused

DEBUG => true_value

generateID

debug

Chapter 6

Configuration

You have only to set the switchs *RBAC* to *(yes/no/on/off)* in all conf-files of the modules.

Part V

Apendices

Chapter 7

Glossar

ACL	AccessControlList
CA	Certification Authority
CRL	Certificate Revocation List
CRR	Certificate Revocation Request
DBI	databaseinterface of Perl
DER	binary format for PKI-files
OpenCA::ACL	
OpenCA::DB	
OpenCA::DBI	
OpenCA::DBIS	
OpenCA::RBAC	
OpenCA::ROLE	
OpenCA::SYNC	
PEM	text format for PKI-files
PKI	Public Key Infrastructure
Public	the name of the webgateway of OpenCA
RA	Registration Authority
RAServer	the server which the RA(O)s use
RAO	RA Operator (works in the RA)
RBAC	RoleBasedAccessControl
RDBMS	RelationalDataBaseManagementSystem
SQL	Structured Query Language

Chapter 8

Standardsoftware used by OpenCA

Chapter 9

About the authors

This chapter is mainly there to give you an overview about who has written a part so that you contact the right one if you need help or you think the author need it ;-D

9.1 Michael Bell

- OpenCA::DBI
- OpenCA::DBIS

Chapter 10

Contact

If you find anything what is not correct or you have written some documentation which should be included please contact the authors of the module or one of the projectmanagers or writers.