InSiTo Library Architecture Documentation

Version 1.1

Falko Strenzke, FlexSecure GmbH
strenzke@flexsecure.de

September 2nd, 2008

Contents

1 Introduction

This document gives an overview of the design of the InSiTo version [3] of the Botan cryptographic library [2]. The basic design of the InSiTo version of the library is equal to the one of Botan, thus, to a certain extend, this document can be used as an architectural reference for Botan as well.

The structure of this document is as follows. Each of the subsequent sections covers a certain functionality domain. Each of these domains is assigned a group of header and source files which implement the respective functionality. Where appropriate, the functions of objects contained in those files are explained. The order in which the files are listed is according to their logical dependencies in most cases, but sometimes an alphabetic order is chosen. This is the case when no true dependencies exist between the files. In some sections, extra subsections provide the explanation of important objects, functions and concepts.

2 Source Code Structure

The libraries source code is divided among the following directories.

- `include` – the library headers. The `include` directory holds two subdirectories: the `api/` directory contains those headers that are supposed to be used by clients, the `int/` directory contains library internals.
- `src` – the library source files.
- `modules` – platform specific code is contained in this directory.

While there is not much to say about the general buildup of the first two directories, the modules directory deserves some more attention. The platform specific code in this directory is used for the following functionalities:

- assembler support for certain symmetric algorithms (`alg_amd64` and `alg_ia32`).
- entropy sources based on OS specific APIs and command (`es_...`)
- enabling to use Unix file descriptors with the libraries pipe concept (see Section 6) (`fd_unix`).
- memory locking functionality, i.e. enabling to keep memory from being swapped out to the swap file (`ml_...`).
- assembler support for low level integer operations (`mp_...`).
- pthread support for different OS', only needed for the unit tests (`mt_...`).
- mutexes in order to enable thread-safety on all platforms (`mux_...`).
- timer functionality (`tm_...`).

- support for usage of either the platforms standard libraries tr1/memory functionality or the one in a boost version (`tr1_inclusion`). Which one will be used can be toggled in the CMake frontend (see the User Manual).

The only non platform specific code is found in the directories `comp_bzip2` and `comp_zlib` which are offered as alternatives here.

At compile time it is determined which modules will be included on which platform. This is based on the evaluation of the `CMakeLists.txt` and `CMakeLists<...>.txt` files in the `modules` directory.

There is basically one file not found in the `modules` directory which is also influenced at compile time. This is the file `modules.cpp`, which defines the member functions of the `Builtin_Modules`. The related header file is `int/modules.h`. `modules.cpp` contains preprocessor macros that ensure that the member functions of the `Builtin_Modules` class return exactly those classes from the `modules` directory that are enabled by the build system. It is possible to use the static interface defined in `modules.h` because all variable classes are treated polymorphically.

3 Global State

This section describes the functionality related to the libraries global state. The library's global state features functionalities like the global RNG, the library configuration, and the memory allocation system of the library.

3.1 Related Header and Source Files

- `api/config.h`, `config.cpp`, , `inifile.cpp` – offer functionality related to the global library configuration. This includes functions for parsing configuration files, setting up the default configuration, or accessing configuration.

- `api/init.h`, `init_def.cpp`, `init_opt.cpp` – contain the classes `LibraryInitializer` and `InitializerOptions`. The library initializer is the object that sets up the global state in its constructor, and destroys it in its destructor. For this reason, the library initializer object has to be the last object of all library objects to be destructed.

- `int/libstate.h`, `libstate.cpp` – the global `Library_State` object itself. It manages the following members:

  - the global configuration (See [4] for details about the global library configuration.)
  - a mutex factory
  - lists for mutex locks
  - a timer object

  - the memory allocators (Refer to Section 5 for information about the libraries memory allocation system.)
  - a pointer to a user interface
  - a character transcoder
  - the global random number generator (RNG)
  - a list of entropy sources (For information about the role of the global RNG and the entropy sources refer to Section 10.)
  - a list of the available cryptographic engines (See Section 7 for the information about the engine concept.)

- `policy.cpp` – contains the default configuration, i.e. the one used when no configuration file is specified.

4 Type System, Smart Pointers

This section deals with the Botan specific types as well as the smart pointers that are widely used in the InSiTo version.

4.1 Related Header and Source Files

- `api/freestore.h` – enforces the use of smart pointers for all `Filter` objects. Refer to the API documentation for details.
- `api/types.h` – defines type aliases for integer types.
- `api/enums.h` – defines various enumerations used throughout the library.

4.2 Important Objects, Functions and Concepts

4.2.1 Smart Pointers

The types of smart pointers used in the library are `std::auto_ptr` and `std::tr1::shared_ptr` . If the latter is not provided by the platforms standard library, the boost shared pointers will be used. The concept of the usage of either type of smart pointer is the following. Every time, when a library function returns an independent object, i.e. a newly created object or a copy of an existing object, `auto_ptr`s are used. In all cases where the library function stores the object itself after it returned the pointer to the caller, a `shared_ptr` is used.

4.2.2 `SharedPtrConverter`

`SharedPtrConverter`s are used as parameter types in function declarations in order to allow the client to conveniently pass either a plain pointer, `auto_ptr` or `shared_ptr`.

5 Memory Allocation System

The memory allocation in the library is handled by the global state (see Section 3.1), which holds multiple allocator objects to this end.

5.1 Related Header and Source Files

- `int/allocate.h` – defines abstract `Allocator` objects.
- `int/mem_pool.h`, `mem_pool.cpp` – defines abstract `Pooling_Allocator` objects, derived from `Allocator`.
- `int/defalloc.h`, `defalloc.cpp` – define the classes `Malloc_Allocator` and `Locking_Allocator`, both being derived from `Pooling_Allocator`. While the former is the normal allocator, the second one is responsible for allocation of so called secure memory, which will never be swapped out by the operating system, i.e. never be written to disk. Note that restrictions with respect to this property of the secure memory concept apply to certain platforms, see below.
- `api/secmem.h` – defines a number of buffer classes, that differ in the properties of having fixed or variable length, and making use of secure memory or not. Refer to the API documentation for details.
- In the `modules` directory, the low level system calls for memory locking are implemented.
  - `ml_fail/mlock.cpp`
    This is the default implementation which is used on all platforms which do not support memory locking. It will cause all attempts to create objects using secure memory to fail. A `SecureMemory_Failure` will be thrown in these cases.
  - `ml_unix/mlock.cpp`
    Unix systems support memory locking in an explicit and clearly documented way. This allows a proper implementation on these platforms. The system call `mmap()` is used to allocate memory for the process and afterwards it will be locked by the help of the system call `mlock()`.
  - `ml_win32/mlock.cpp`
    With Windows, the situation is rather unclear. The system calls that are used are `VirtualAlloc()` and `VirtualLock()`. While the former is the Windows counterpart of `mmap()`, the latter does not really seem to be a true equivalent to `mlock()`. While the MSDN documentation states that it actually locks the specified address range into RAM[1],

[1] `http://msdn.microsoft.com/en-us/library/aa366895(VS.85).aspx` (5th, Aug., 2008)

other sources state otherwise[2]. Thus using MS Windows the full security as provided by Unix cannot be claimed.

- `mem_ops.h` – wrappers for the C standard library memory related functions.

5.2 Important Objects, Functions and Concepts

The allocation system basically works as follows. When the `LibraryInitializer` is created, the passed `InitializerOptions` determine whether secure memory will be used at all. To set the initialization options please refer to the API documentation of `InitializerOptions`. If secure memory is used, those classes defined in `secmem.h`, that are labeled "secure", will make use of secure memory.

In any case, upon the creation of any `secmem.h`-object, this object will initially retrieve an `Allocator`-object from the global `Library_State` object. If secure memory is globally enabled (see the paragraph above), and the object is of the "secure" kind, the `Allocator`-object will be of the type `Locking_Allocator`, otherwise it will be a `Malloc_Allocator`. The `Allocator`-object will be used to allocate the "secure" objects internal buffer memory.

In case of the `Malloc_Allocator` allocation will take place via the `malloc()` function. The steps performed by a `Locking_Allocator` are more complicated. Initially, it has to be assured that the memory was allocated page-wise. This is essential because both the Unix and Windows memory locking functions work page-wise, i.e. they lock all pages in the specified address range. If a chunk A of allocated memory would not occupy a full multiple of pages, another chunk B of secure memory might be allocated on one of these pages too. The unlocking of B, which alway precedes the deallocation, would unintentionally also unlock A. To prevent this, the page size is determined via corresponding system calls and multiples of pages aligned on page borders are allocated. This is ensured centrally by the `Locking_Allocator`, which only leaves the platform specific system calls to the functions in the afore mentioned files in the modules directory.

Upon a deallocation request, the pages are initially overwritten with zero bytes, then unlocked via the respective system call, and finally unmapped.

6 Pipe Concept

The pipe concept allows for the convenient handling of data streams and cryptographic algorithms. The basic idea is that the client creates so called pipes, which are quiet equivalent to the Unix pipes. Each pipe can then be filled with

[2] `http://blogs.msdn.com/oldnewthing/archive/2007/11/06/5924058.aspx` (5th, Aug., 2008)

a sequence of filters. The usage of Pipes and filters is described in the API Documentation and the user manual[4].

The data source concept is associated with the pipe concept, because `Pipe`s are implementations of `DataSource`s. A `DataSource` can be used to create CVC or X509 objects, for instance. Figure 1 shows the inheritance hierarchy for `DataSource`.

Figure 1: Inheritance graph for `Data_Source`

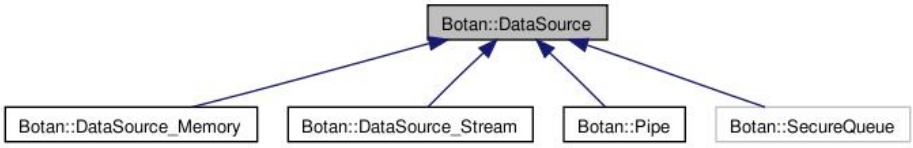


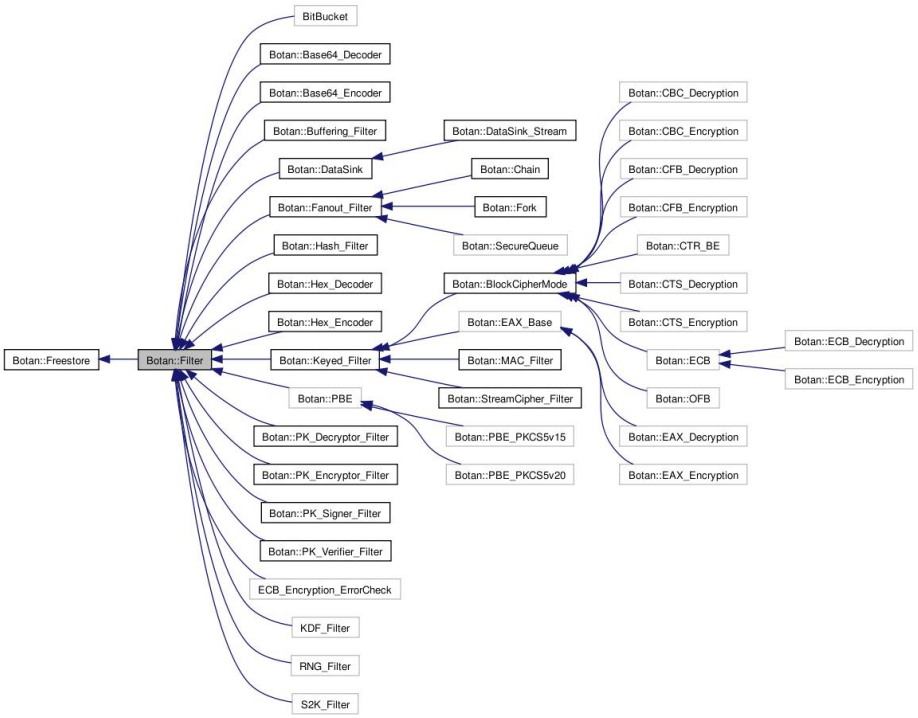
Figure 2 shows the various classes derived from `Filter`.

6.1 Related Header and Source Files

In the following, those header and source files related to the pipe concept are listed. In most cases no description is given, because the details about the classes contained in these files are well documented in the API Documentation.

- `api/base64.h`, `base64.cpp`
- `api/basefilt.h`, `basefilt.cpp`
- `api/buf_filt.h`, `buf_filt.cpp`
- `api/data_snk.h`, `data_snk.cpp`
- `api/data_src.h`, `data_src.cpp`
- `api/filter.h`, `filter.cpp` – among other things, this file defines the `Filter` base class.
- `api/filters.h`, `filters.cpp`
- `api/hex.h`, `hex.cpp`
- `api/pipe.h`, `pipe.cpp`, `pipe_io.cpp`, `pipe_rw.cpp` – defines the `Pipe` class.
- `api/pk_filts.h`, `pk_filts.cpp` – defines various public key related filters.
- `api/data_src.h`, `data_src.cpp` – define the abstract `DataSource` base class, and the derived classes `DataSource_Memory` and `DataSource_Stream`

Figure 2: Inheritance graph for `Filter`



- `int/secqueue.h`, `secqueue.cpp` – define the class `SecureQueue` which is used internally by the library.
- `int/out_buf.h`, `out_buf.cpp` – define the `Output_Buffer` class used internally by the library.

7 Engine Concept

This section covers the engine concept provided by the library. The basic idea is that it is possible to include various cryptographic engines in the library, e.g. it would be possible to use the OpenSSL engine, thus making it possible for the library to use the OpenSSL algorithms. However, in order to fulfill the desired security level, no additional engines are configured in the InSiTo version, only the default Botan engine is available.

A Botan cryptographic engine has member functions that return cryptographic operations. These operation objects are used in the implementation of the public key algorithms. There, the collaboration hierarchy is as follows. A public key object contains a core object, this object in turn holds an opera-

tion which features the actual implementation of the public key algorithms. An example would be the function

```
std::tr1::shared_ptr<ECDSA_Operation>
  ecdsa_op(EC_Domain_Params const& dom_pars,
     Botan::math::BigInt const& priv_key,
     Botan::math::ec::PointGFp const& pub_key) const;
```

which can be found in the Botan default engine in the file `int/eng_def.h`. The returned operation will be used inside an `ECDSA_Core`, which in turn is contained in an `ECDSA_PublicKey` or `ECDSA_PrivateKey`. The basic difference between public keys and private keys in this respect is that the public key's operations are lacking the private value, and thus cannot perform the private operation.

While all public key algorithms are implemented as described above, symmetric ciphers are not implemented in terms of cores and operations. The corresponding member functions of the engine return an instance of the symmetric algorithm directly.

However, application code never has to deal with the engine directly. There exist factory / lookup functions for all cryptographic algorithms. Refer to [4] for examples.

7.1 Related Header and Source Files

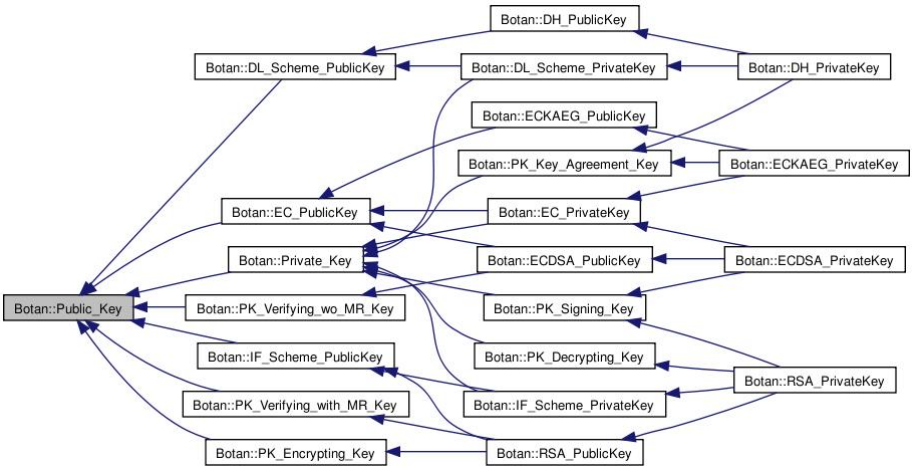
We only assign the core engine files to this section. The files containing the cores and operations for public key algorithms mentioned above are listed in Section 8.

- `int/engine.h` – defines the abstract `Engine` base class.
- `int/eng_base.cpp` – implementation of the `Engine` class.
- `engine.cpp` – defines global functions for retrieving certain algorithms.
- `int/eng_def.h` – definition of the libraries `Default_Engine` which is used if no other engine is configured by the client.
- `int/def_alg.cpp`, `def_mode.cpp` – implementation of the `Default_Engine`.

8 Public Key Algorithms

This section deals with the realization of the public key algorithms in Botan. The basic concept for the realization of the public key schemes is that most functionality is packed into the key objects. Each public key scheme features a public key, and a private key which is derived from this key. The full inheritance graph for public and private keys is depicted in Figure 3.

Figure 3: Inheritance graph for `Public_Key`



8.1 Related Header and Source Files

8.1.1 Key Objects

- `api/pk_keys.h`, `pk_keys.cpp` – define the abstract public key base classes. All public and private key classes are derived from `Public_Key` and `Private_Key` defined in `pk_keys.h`.
- `api/dl_group.h`, `dl_group.cpp` – define the class `DL_Group`.
- `api/dl_algo.h`, `dl_algo.cpp` – define the abstract discrete logarithm key base classes.
- `api/dh.h`, `dh.cpp` – define the Diffie-Hellman keys, derived from the discrete logarithm base class.
- `api/ec.h`, `ec.cpp` – define the ECDSA and ECKAEG key classes.
- `api/ec_dompar.h`, `ec_dompar.cpp` – define the elliptic curve domain parameters (`EC_Domain_Params`).
- `int/ecdsa.h`, `ecdsa.cpp` – defines the `ECDSA_Signature` class used inside the library.
- `int/if_algo.h`, `if_algo.cpp` – contain the base classes for the integer factorization (IF) based public key scheme keys.
- `api/rsa.h`, `rsa.cpp` – define RSA public and private keys. These are derived from the IF key base classes defined in `if_algo.h`.

- `api/pubkey.h`, `pubkey.cpp` – define various encryptor, decryptor, signer, and verifier classes. Refer to the API Documentation for details on these classes.
- `int/pk_util.h`, `pk_util.cpp` – contain base classes for EME, EMSA, KDF, and MGF classes.
- `int/pk_algs.h`, `pk_algs.cpp` – factory functions to create public and private keys.
- `api/keypair.h`, `keypair.cpp` – functions for checking public key scheme key pairs.
- `int/alg_id.h`, `alg_id.cpp` – define the `AlgorithmIdentifier` class.
- `api/pkcs8.h`, `pkcs8.cpp` – define the abstract base class for abstract PKCS#8 encoders and decoders. Each type of private key defines an encoder and decoder derived from these base classes (as inner classes).

8.1.2 Algorithm Implementations

- `int/pk_ops.h` – defines abstract public key operations (for the meaning of a public key operation, refer to Section 7).
- `int/def_ops.cpp` – definition and implementation of the default public key operations. These are the operations that are returned by the libraries default engine (see Section 7).
- `int/pk_core.h`, `pk_core.cpp` – define the public key cores (for the meaning of a public key core, refer to Section 7).
- `int/blinding.h`, `blinding.cpp` – realizes a blinding countermeasure against side channel attacks on integer factorization based public key algorithms.

8.1.3 Encoding

- `int/emsa.h`, `emsa1.cpp`, `emsa1_bsi.cpp`, `emsa2.cpp`, `emsa3.cpp`, `emsa4.cpp`, `emsa_raw.cpp` – the message-encoding methods for signature schemes with appendix.
- `int/eme.h`, `eme1.cpp`, `eme_pkcs.cpp` – the message-encoding methods for encryption schemes.

8.1.4 Factory Functions

- `api/look_pk.h`, `look_pk.cpp` – factory functions for various public key filter objects.

8.2 Side Channel Attack Countermeasures

The following side channel attack countermeasures are implemented in the library:

- Base Blinding for RSA and DH Algorithms. This is a valid countermeasure against timing attacks. The blinding is implemented in terms of the class `Blinder`, defined in `int/blinding.h`. The blinding is realized in the file `pk_core.cpp` for the RSA and DH algorithms.
- Add-and-double-always and Exponent Blinding for ECDSA and ECKAEG. Refer to [5] for details about the EC countermeasures.

All these countermeasures are valid to defeat timing attacks. Although the implementation will be definitely more resistant against power analysis attacks, to get a qualified statement one has to perform an evaluation of the countermeasures on the respective platform. The countermeasures will not defeat microarchitectural attacks such as cache attacks and branch prediction attacks.

9 Symmetric Algorithms

The library distinguishes between the following types of symmetric algorithms which are represented by the following abstract base classes:

- `BlockCipher`
- `HashFunction`
- `MessageAuthenticationCode`
- `StreamCipher`[3]

All of these classes are defined in `int/base.h` and `base.cpp`. The modes of the block ciphers are implemented as `Keyed_Filter`s. In this way, the client code can use the convenient pipe/filter API for these algorithms. See Figure 2 for details about the inheritance hierarchy of `Filter` and the symmetric algorithms.

9.1 Related Header and Source Files

9.1.1 Symmetric Algorithms in general

- `api/lookup.h`, `get_algo.cpp`, `get_enc.cpp` – factory functions for various symmetric algorithms.

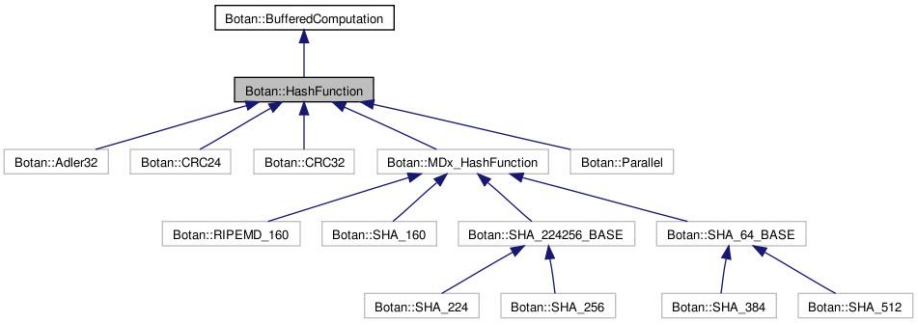
[3] In the InSiTo version, no stream cipher algorithm is implemented.

9.1.2 Hash Algorithms and Checksums

Note that all checksum algorithms are realized as hash functions in the library. Figure 4 gives an overview over the inheritance hierarchy of the hash function classes.

- `int/mdx_hash.h`, `mdx_hash.cpp` – abstract base class for message digest hash algorithms.
- `int/adler32.h`, `adler32.cpp` – the `Adler32` checksum algorithm. Is derived from `HashFunction`.
- `int/crc24.h`, `crc24.cpp` – cyclic redundancy check (CRC) checksum.
- `int/crc32.h`, `crc32.cpp` – cyclic redundancy check (CRC) checksum.
- `int/sha160.h`, `sha160.cpp` – the SHA-1 hash algorithm.
- `int/sha256.h`, `sha256.cpp` – the SHA-224 and -256 algorithms.
- `int/sha_64.h`, `sha_64.cpp` – the SHA-385 and -512 algorithms.
- `int/rmd160.h`, `rmd160.cpp` – the RIPEMD-160 hash algorithm.
- `int/par_hash`, `par_hash.cpp` – a wrapper for hash function to enable the processing of data by different hash functions in parallel.

Figure 4: Inheritance graph for `HashFunction`



9.1.3 Block Ciphers

- `int/aes.h`, `aes.cpp`, `aes_tab.cpp` – the AES encryption algorithm.
- `int/des.h`, `des.cpp`, `des_tab.cpp` – the `DES`, `TripleDES` and `DESX` encryption algorithms.

9.1.4 Block Cipher Modes

Note that unless stated otherwise, for each block cipher mode an encryption and a decryption `BlockCipherMode` is implemented.

- `api/modebase.h`, `modebase.cpp` – define the class `BlockCipherMode`.
- `int/cbc.h`, `cbc.cpp` – CBC encryption and decryption.
- `int/cfb.h`, `cfb.h` – CFB encryption and decryption.
- `int/ctr.h`, `ctr.cpp` – CTR block cipher mode. Here, no distinct `BlockCipherMode`s exist for encryption and decryption.
- `int/cts.h`, `cts.cpp` – CTS encryption and decryption.
- `int/eax.h`, `eax.cpp` – EAX authenticated encryption mode.
- `int/ecb.h`, `ecb.cpp` – ECB encryption and decryption.
- `int/ofb.h`, `ofb.cpp` – OFB block cipher mode. Here, no distinct `BlockCipherMode` exist for encryption and decryption.
- `int/mode_pad.h`, `mode_pad.cpp` – define various block cipher padding modes.

9.1.5 MACs

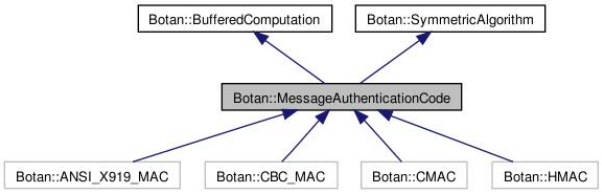
Like block ciphers, MACs are implemented as `Keyed_Filter`s. Figure 5 shows the inheritance graph for `MessageAuthenticationCodes`.

- `int/cbc_mac.h`, `cbc_mac.cpp` – CBC-MAC algorithm.
- `int/cmac.h`, `cmac.cpp` – CMAC algorithm.
- `int/hmac.h`, `hmac.cpp` – HMAC algorithm.
- `int/x919_mac.h`, `x919_mac.cpp` – the ANSI X919 MAC.
- `int/hash_id`, `hash_id.cpp` – functions related to hash IDs.

9.1.6 Other Symmetric Algorithms

- `int/mgf1.h`, `mgf1.cpp` – mask generation function.
- `int/kdf.h`, `kdf.cpp` – key derivation functions.
- `api/pbe.h`, `get_pbe.cpp` – Password based encryption (PBE) functionality abstract base class and lookup functions.
- `api/pkcs5.h`, `pkcs5.cpp` – PKCS#5 implementation.
- `api/pbe_pkcs5.h`, `pbes1.cpp`, `pbes2.cpp` – PKCS#5 PBE implementation.
- `api/s2k.h`, `s2k.cpp` – string to key functionality abstract base class.

Figure 5: Inheritance graph for `MessageAuthenticationCodes`



10 Random Number Generators and Entropy Sources

The library realizes random number generators (RNGs) as `RNG` objects. They are implemented as pseudo random number generators which are initially seeded from the libraries entropy sources, so that they actually function as RNGS.

The global RNG is held by the library state (see Section 3.1). It will be used by all cryptographic algorithms when they need random numbers, e.g. the key generation algorithms for public key schemes. By default, the global RNG will be of type `ANSI_X931_RNG`, this is specified in the file `init_def.cpp`, inside the function

```
void LibraryInitializer::initialize(
                        const InitializerOptions& args,
                        Modules& modules)
```

Note that the global RNG is seeded intially during the library initialization, i.e. in the constructor of the library initializer. This also happens inside the function `initialize()` mentioned above. Here, the boolean parameter `slowpoll` in the call to`LibraryState::seed_prng()` is specified as true. This means that the entropy sources that are used will perform a rather slow operation which yields a larger amount of entropy than a so called fast poll, which is also offered by all `EntropySource` implementations.

10.1 Related Header and Source Files

- `api/base.h`, `base.cpp` – define the abstract base class `RandomNumberGenerator` and `EntropySource`.
- `int/bbs.hm` `bbs.cpp` – the implementation of the Blum-Blum-Shub PRNG.
- `int/sha1prng.h`, `sha1prng.cpp` – the implementation of the `SHA1PRNG`, according to FIPS PUB 186-2, Appendix 3.1.
- `int/x931_rng.h`, `x931_rng.cpp` – `ANSI_X931_RNG`, the PRNG defined in ANSI X9.31, Appendix A.

- `api/rng.h`, `rng.cpp` – access functions for the global RNG.
- `int/buf_es.h`, `buf_es.cpp` – define the `BufferedEntropySource` class.
- `int/es_file.h`, `es_file.cpp` – define an entropy source that retrieves entropy from a file.
- `int/randpool.h`, `randpool.cpp` – define the `Randpool` RNG.
- in the `modules` directory, the following OS specific entropy sources are defined.
  - `es_beos/`: `es_beos.h`, `es_beos.cpp` – a BeOS specific entropy source which is derived from `BufferedEntropySource`.
  - `es_capi`: `es_capi.h`, `es_capi.cpp` – a Windows 32-bit specific entropy source, derived directly from `EntropySource`.
  - `es_egd`: `es_egd.h`, `es_egd.cpp` – an entropy source internally using the Entropy Gathering Daemon (EGD) on Unix systems.
  - `es_unix`: `es_unix.h`, `es_unix.cpp`, `unix_src.cpp` – a Unix specific entropy source that draws entropy from the results of various Unix commands, which depend on the current system state. It is derived from `Buffered_EntropySource`.
  - `es_win32`: `es_win32.h`, `es_win32.cpp` – a Windows 32-bit specific entropy source derived from `Buffered_EntropySource`. It uses various system state information sources like the cursor position, current memory status and the OS' performance counter.

11 Arithmetic

The library internally uses and makes available three types of arithmetics to client code:

- arbitrary precision integer arithmetic (APIA),
- modular arithmetic (MA), and
- elliptic curve arithmetic (ECA).

There exists a clear collaboration hierarchy: the ECA builds upon the MA, which in turn is founded on the APIA. All three arithmetics are well documented in the API Documentation.

11.1 Related Header and Source Files

In the following, the header and source files, which define and implement the low level functionality of the APIA will not be listed.

- api/mathbigint.h, bigint.cpp, bigint_code.cpp. bigint_io.cpp, bigint_ops3.cpp, bigint_rand.cpp – definition and implementation of the BigInt class, which realizes APIA.
- int/math/bigint/mp_types.h – defines the word size used in the low level APIA routines. The word size is chosen platform dependent, this happens at compile time.
- int/math/bigintfuncs.h, bigintfuncs.cpp – a number of utility and number theoretic functions related to the APIA and MA.
- api/math/gf/gfp_element.h, gfp_element.cpp – defines and implements GF(p) (GFpElement) elements, i.e. the MA.
- api/math/ec/gfp_modulus.h, gfp_modulus.h – define the class GFpModulus, which encapsulates the modulus and modulus related values for the use in GFpElement.
- api/math/ec/point_gfp.h, point_gfp.cpp – the elliptic curve point class PointGFp. This class realizes the ECA.
- api/math/ec/curve_gfp.h, curve_gfp.cpp – define the class CurveGFp.

12 ASN1

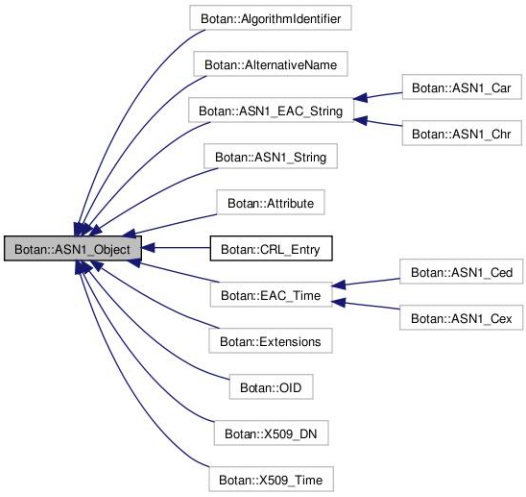
The library features ASN1 DER and PEM encoding and decoding functionality. This functionality is used for the supported CVC and X509 objects (see Sections 13 and 14).

The ability of an object to be encoded in DER format is achieved by inheriting from ASN1_Object, defined in asn1_int.h. The inheritance hierarchy of ASN1_Object is depicted in Figure 6. This includes classes that belong to the CVC and X509 functionality.

12.1 Related Header and Source Files

- int/ber_dec.h, ber_dec.cpp – a BER decoder.
- int/der_enc.h, der_enc.cpp – a DER encoder.
- int/asn1_int.h, asn1_int.cpp – define the abstract ASN1_Object base class along with some other ASN1 related classes.
- int/asn1_obj.h, api/asn1_oid.h, asn1_alt.cpp, asn1_att.cpp, asn1_dn.cpp, asn1_eac_str.cpp, asn1_eac_tm.cpp, asn1_int.cpp, asn1_ku.cpp, asn1_oid.cpp, asn1_str.cpp, asn1_tm.cpp – define various ASN1 objects, i.e. classes derived from ASN1_Object.
- int/pem.h, pem.cpp – core functions for PEM encoding.

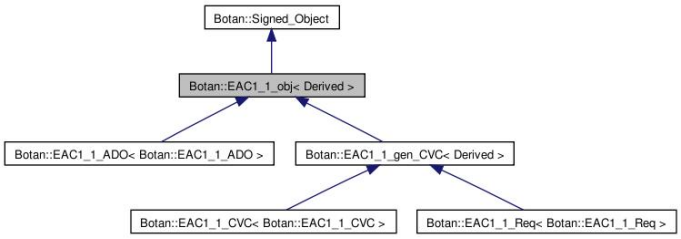
Figure 6: Inheritance graph for `ASN1_Object`



13 CVC

Card verifiable certificates according to EAC 1.1 are realized by the inheritance hierarchy depicted in Figure 7.

Figure 7: Inheritance graph for `EAC1_1_obj`



13.1 Related Header and Source Files

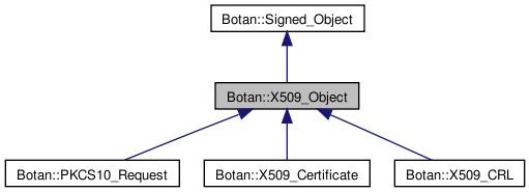
- `api/eac_obj.h` – defines the abstract base class for all EAC1_1 objects. It is derived from `Signed_Object` which is also the base class for all X509 signed objects.
- `api/cvc_ado.h`, `cvc_ado.cpp` – define the class `EAC1_1_ADO`.
- `api/cvc_gen_cert` – define the abstract base class for both CVC requests and certificates (see below). This class holds the functionality that is shared among both derived classes.

- `api/cvc_cert.h`, `cvc_cert.cpp` – define the class `EAC1_1_CVC`, i.e. the actual certificates.
- `api/cvc_req.h`, `cvc_req.cpp` – define the class `EAC1_1_Req` which represents CVC requests.
- `api/cvc_ca.h`, `cvc_ca.cpp` – define a CA class that has one static function which enables the creation of CVCs.
- `api/cvc_self.h`, `cvc_self.cpp` – contain a number of convenience functions to produce certificates from certificate requests.
- `api/cvc_key.h` – define the abstract public key encoder and decoder for CVCs. Both must be implemented for each public key class that is supposed to work with CVCs.

14 X509

The library supports X509 Certificates, PKCS#10-Requests and CRLs. The inheritance hierarchy is shown in Figure 8. All classes belonging to this scope are well documented in the API Documentation.

Figure 8: Inheritance graph for `X509_Object`



14.1 Related Header and Source Files

- `api/x509_obj.h`, `x509_obj.cpp` – abstract base class `X509_Object` for all X509 objects.
- `api/x509_ca.h`, `x509_ca.cpp` – define the `X509_CA` class.
- `api/x509_crl.h`, `x509_crl.cpp` – define the `X509_CRL` class.
- `int/crl_ent.h`, `crl_ent.cpp` – define the `CRL_Entry` class.
- `api/x509_key.h`, `x509_key.cpp` – abstract base classes for X509 public key encoders and decoders. As in the case of CVC, each public key class has to define derived classes of those base classes in order to support X509 certificates.

- api/x509cert.h, x509cert.cpp – X509_Certificate class.
- api/x509self.h, x509self.cpp, x509opt.cpp – define the class X509_Cert_Options and functions for the creation of X509 objects.
- int/x509_ext.h, x509_ext.cpp – contains various classes that represent X509 certificate extensions.
- int/x509stat.h, x509stat.cpp – define the X509_GlobalState class which manages the default set of X509 certificate extensions.
- api/pkcs10.h, pkcs10.cpp – define the class PKCS10_Request.
- int/certstor.h, certstor.cpp – define an abstract certificate store class.
- int/x509stor.h, x509stor.cpp – define a X509 certificate store used in X509 self test.
- int/x509find.h, x509.cpp – define search algorithms for the above X509 certificate store.

15 Miscellaneous and Utility

In this section, all files are covered that do not fit into any of the other sections.

15.1 Related Header and Source Files

- api/botan.h – this file simply includes other generally needed header files of the library.
- int/charset.h, charset.cpp – contain a character set transcoder abstract base class used by the library.
- int/def_char.h, def_char.cpp – the libraries default character set transcoder.
- int/datastor.h, datastor.h – define a DataStore class used inside the library, for instance by the X509 certificate functionality.
- int/dsa_gen.cpp – provides functions used by certain public key classes.
- api/exceptn.h, exceptn.cpp – define the libraries exception classes.
- int/fips140.h, fips140.cpp – define functions for performing a self test, which can be triggered at the library initialization when the LibraryInitializer is configured accordingly.
- int/loadstor.h – defines byte manipulation functions.

- `int/mutex.h`, `mutex.cpp` – define mutex objects for thread safety. Platform specific implementation are found in the `modules` directory.
- `api/oids.h`, `oids.cpp` – define lookup functions for OIDs.
- `int/parsing.h`, `parsing.cpp` – define utility functions for string parsing used internally by the library.
- `api/signed_obj.h`, `signed_obj.cpp` – define `Signed_Object`, the base class for all CVC and X509 objects.
- `int/stl_util.h` – the Standard Template Library related utility functions used in the library.
- `int/symkey.h`, `symkey.cpp` – define an `OctetString` class.
- `int/ta.h`, `ta.cpp` – maintain global counters that are used for side channel resistance tests of EC algorithms. Counters are only affected if corresponding compiler flags are enabled. Refer to [5].
- `int/timers.h`, `timers.cpp` – define time related utility functions.
- `int/ui.h`, `ui.cpp` – define a default user interface for password dialogues.
- `int/util.h`, `util.cpp` – define utility functions used internally by the library.
- `api/version.h`, `version.cpp` – define functions for retrieving the library version.

References

[1] Technical Guideline TR-03111 “Elliptic Curce Cryptography Based on ISO 15946”, Version 1.00, 14.02.2007, Bundesamt fr Sicherheit in der Informationstechnik

[2] `http://botan.randombit.net/`

[3] `http://www.flexsecure.eu/insito/index.html`

[4] InSiTo Library User Manual

[5] Falko Strenzke und Patrick Sona: “AP5 - QS/Tests - Dokumentation InSiTo-Bibliothek, Version 1.4”, 2008

Index