

NetCrypto, NetPKI and NetTLS an OpenSSL replacement

Richard Levitte <richard@levitte.org>
OpenSSL developer

September 28, 2004

Abstract

OpenSSL is a well known package of crypto functions, PKI routines and an SSL/TLS protocol family implementation, with some useful generic functionality thrown in. However, the way it and its predecessor – `SSLey` – has been implemented leaves little room for change without throwing havoc among the users and application writers. I am about to describe my thoughts about a new set of packages that reimplement all of `OpenSSL`, how backward binary compatibility will be assured (to the furthest possible extent) and in what way it will be easy to extend. When it comes to the first package – `NetCrypto` – I’m talking out of the experience I had implementing it.

Part I The idea

1 Introduction

OpenSSL is a well known package of crypto functions, PKI routines and an SSL/TLS protocol family implementation, with some useful generic functionality thrown in. For what it’s worth, it’s a rather great package that solves many of the problems other implementors would otherwise face, by being free for anyone to use free of charge and by being loaded with features that any security/crypto software might want.

However, the way it¹ has been implemented leaves little room for change without throwing havoc among the users and application writers, in form of binary incompatibilities with previous versions and the necessity to make changes in the source any time there’s a change in `OpenSSL` other than small bugs in the routines themselves. API changes and changes in publicly declared structures are commonplace and should be expected whenever x or y changes in the version number, $0.x.y$. This has been very confusing, and has spurred irritation among some hackers and threats of a fork that “does it right”.

Also, in recent versions of `OpenSSL`, a possible chicken and egg problem may appear. Starting with version 0.9.7, it’s possible to have it use Kerberos 5 tickets for authentication in the TLS protocol. As it is now, the only Kerberos 5 implementation this works with is MIT Kerberos. However, there’s already code in place to at least try to use Heimdal instead, and that’s where the fun begins.

¹Its predecessor, `SSLey`, really. `OpenSSL` carries a lot of legacy from there, and will probably keep doing so for quite a while longer

Heimdal² uses `libcrypto`³ to get crypto routines. `libssl`⁴ would in turn use Heimdal to get Kerberos 5 routines. Since all of OpenSSL is built in one go and there's no (easy) way to build it one part at a time, the question is what to build first, OpenSSL or Heimdal? This is actually quite crucial for some package builders, like the ports (or whatever they are called) directories in various BSD implementations.

Furthermore, some OpenSSL routines should really not be publicly available (or at least not declared publicly). It's been recommended to use the EVP layer to access crypto routines for ages, but program implementors still use the raw crypto routines (which vary a bit between crypto algorithms) and make their own umbrella layer on top of them, thus basically parallelling the EVP layer.

To boot, the use of some crypto algorithms are not allowed in some countries, or are restricted by patents and costly⁵ license schemes. The solution in OpenSSL is to allow the configuration process to restrict compilation of some parts, by using C macros called `OPENSSL_NO_name`, where *name* is the name of the algorithm in question. The problem with this is that the way to use specific algorithms without getting more or less all of `libcrypto`⁶ is to call very specific EVP routines, and if the requested algorithm isn't built as part of OpenSSL, the corresponding EVP routine to fetch the algorithm isn't available either. That's not so hard as long as one keeps to static linking (because the configured `OPENSSL_NO_name` macros are defined in `openssl/opensslconf.h`), and as long as you keep track of them all. Enter dynamic libraries (`libcrypto.so`)...

2 The overall solution

First of all, the chicken and the egg problem presented in the introduction is quite easily solved by creating three different packages that reimplement the three main parts of OpenSSL. For that purpose, I have "invented" the packages are `NetCrypto`, `NetPKI` and `NetTLS`. `NetCrypto` implements a crypto algorithm core and a few algorithm classes, `NetPKI` will implement PKI functionality, such as X.509 certificates, ASN.1, some PKCS modules and so on and so forth, and `NetTLS` will implement the SSL/TLS protocol family.

Second, this problem with disabled algorithms and such would suggest that algorithms should be made available in a dynamic manner⁷. The solution, which is implemented in `NetCrypto`, is therefore to have a generic algorithm store as the core, and add stuff around that. With a sufficiently generic core, algorithm classes⁸ can easily be defined, as well as class-specific APIs that act as accessors to the internal class functionality and instance data. When algorithm classes are available, it's fairly easy to implement algorithms that can then be used in a uniform manner. A reference to each algorithm implementation is stored in an internal database at runtime. All this is made in such a way that there is only one way to get to an algorithm. Methods to have algorithms and groups of algorithms implemented as separate shared libraries are provided as well⁹.

Third, to avoid all the incompatibility problems (binary as well source), structures are made

²The Kerberos 5 implementation created by a few hackers at the Royal Institute och Technology in Stockholm, Sweden

³The crypto and PKI routine library provided by OpenSSL

⁴The ssl/tls routines library provided by OpenSSL

⁵rather more than less costly

⁶You can actually request an algorithm through a specifier in form of a string. The problem with that is that the routine handling it is bound to all available crypto algorithms, meaning that any use of this routine makes sure the program gets *all* available crypto algorithms linked in. That results in a quite large program...

⁷and certainly not through a routine that links in the whole library

⁸where a class is a numeric identity and a specific set of associated functions

⁹The part implementing this is called ENGINE, mimicking, though only a little, the OpenSSL ENGINE framework

opaque, with very few exceptions¹⁰. Those opaque structures are allocated, handled and freed through the associated functions.

Fourth, there is a stage when an API (an algorithm class, or the code algorithm functions, or the ENGINE functions, ...) reaches maturity. At that point, the API is frozen and declared as written in stone. To help determine the maturity of each part, the following macros should be defined (*packagename* is NC for NetCrypto, NPki for NetPKI and NTLs for NetTLs, *name* is the name of the API):

packagename_name_VERSION

The version number is formed from the year and month of implementation, with the form YYYYMM.

packagename_name_COMPAT

The version number that this package is currently backward compatible with.

packagename_name_STATUS

A keyword saying what the status of the API is. The currently defined statuses are DEVELOPING, TESTING and FROZEN. They are to be used when the current API is being actively developed, when it's under test, and when it reached maturity and is written in stone.

packagename_name_DESCRIPTION

A string describing the API in just a few lines.

Additionally, general utility and compatibility routines that can be used elsewhere have been taken out and placed in a separate library, LPlib.

Part II

Taking a closer look

3 NetCrypto

In the center is the algorithm, a structure to rule them all...
(and information to serve them all)

Everything that can be done with NetCrypto is ruled by the central concept of an algorithm along with the ENGINE framework that allows algorithms to live in separate shared libraries.

The central algorithm structure really only keep track of tuples containing a class identity (a number), an algorithm identity (a string), an ENGINE if the algorithm lives in a separate shared library, class data (function pointers and some class-level data), and init and a cleanup function. This database is governed by functions to register new algorithms and to fetch already existing algorithms, along with a function to create the algorithm structure out of given data.

Algorithm classes are then built around this central structure, without any need to know what the central algorithm structure looks like (there's a silly example in Appendix A). There are a few classes already implemented:

¹⁰the only exception so far is a structure of comparison data used to verify that an algorithm in a shared library can be used with the current implementation of NetCrypto

NC_ALGORITHM_DIGEST

Implements digests. Written in stone, as far as I can see.

NC_ALGORITHM_RAND

Implements random generators, both pseudo and not so pseudo. Written in stone, as far as I can see.

NC_ALGORITHM_CIPHER

Implements symmetric ciphers. Almost written in stone, there's just the question of HSM-stored keys to solve.

NC_ALGORITHM_PKCIPHER

Implements asymmetric ciphers. Still under development.

NC_ALGORITHM_TRANSPORT

Implements transport encoding. Almost written in stone, there's just the need of testing a little more.

Additionally, there's an INFORMATION mechanism, which can be used to save and load keys¹¹, signatures and any other kind of information. This part is currently under development, so I won't discuss it too much.

4 NetPKI

I haven't yet made any specific plans for NetPKI, and it's definitely in the mist right now. I know that I'd like the number of functions to be kept low, and at the same time to have this rich in features and flexibility.

This part will hopefully contain Dr. Stephen Henson's rewrite of the ASN.1 code in OpenSSL. I'm still waiting for his permission.

5 NetTLS

I haven't yet made any specific plans for NetTLS, either. I know that I'd like to keep the number of functions low, and to have it rich in features, as well as easily extensible for new protocol versions. I basically want functionality that incorporates all protocol versions, and where the desired protocol versions are set through options.

Part III

Appendices

A A silly algorithm class

This implements a class that does a one-way inline conversion of bytes, and an example algorithm of this class that makes sure each byte has the parity bit set to even parity.

The converter class (`converter.h` and `converter.c`):

¹¹or references to HSM-stored keys

```

1  /* converter.h */
2
3  #ifndef CONVERTER_H
4  #define CONVERTER_H
5
6  #include <sys/types.h>
7  #include <netcrypto/classes.h>
8  #include <netcrypto/generic.h>
9  #include <netcrypto/engine.h>
10
11 #ifdef __cplusplus
12 extern "C" {
13 #endif
14
15     /* Users MUST define their own class identities with NC_ALGORITHM_USER_BASE
16        as base */
17     #define ALGORITHM_CONVERTER    (NC_ALGORITHM_USER_BASE+1)
18
19     /* Functions that are shortcuts for NC_algorithm and NC_engine_algorithm. */
20     LP_STATUS converter_algorithm(const char *id, const NC_ALGORITHM **res_algo);
21     LP_STATUS engine_converter_algorithm(const char *id, NC_ENGINE *e,
22                                         const NC_ALGORITHM **res_algo);
23
24     /* Functions to create algorithm structures */
25     LP_STATUS
26     create_converter(const char *id,
27                    NC_ENGINE *engine,
28                    LP_STATUS (*convert)(unsigned char *data, size_t datal),
29                    NC_ALGORITHM **res_algo);
30
31     LP_STATUS convert(const NC_ALGORITHM *converter,
32                     unsigned char *data, size_t datal);
33
34 #ifdef __cplusplus
35 }
36 #endif
37
38 #endif

```

```

1  /* converter.c */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <LPmsg.h>
6  #include "converter.h"
7
8  /* This is the private structure that's passed to the generic algorithm
9     creator as class data */
10 typedef struct converter_st
11 {
12     LP_STATUS (*converter)(unsigned char *data, size_t datal);
13 } CONVERTER;
14
15 LP_STATUS converter_algorithm(const char *id, const NC_ALGORITHM **res_algo)
16 {
17     return NC_algorithm(ALGORITHM_CONVERTER, id, res_algo);
18 }
19
20 LP_STATUS engine_converter_algorithm(const char *id, NC_ENGINE *e,
21                                     const NC_ALGORITHM **res_algo)
22 {
23     return NC_engine_algorithm(ALGORITHM_CONVERTER, id, e, res_algo);
24 }
25
26 /* Functions to create algorithm structures */
27 LP_STATUS
28 create_converter(const char *id,

```

```

29         NC_ENGINE *engine,
30         LP_STATUS (*convert)(unsigned char *data, size_t datal),
31         NC_ALGORITHM **res_algo)
32 {
33     CONVERTER *converter = (CONVERTER *)malloc(sizeof(CONVERTER));
34     LP_STATUS status;
35
36     if (converter == NULL)
37         return LP_STS_NOMEM;
38
39     converter->converter = convert;
40
41     status = NC_algorithm_create(ALGORITHM_CONVERTER, id, engine, converter,
42                                NULL, NULL, NULL, res_algo);
43     if (!LP_status_is_OK(status))
44         return status;
45
46     status = NC_algorithm_finalise(*res_algo);
47     return status;
48 }
49
50
51 LP_STATUS convert(const NC_ALGORITHM *converter,
52                 unsigned char *data, size_t datal)
53 {
54     const CONVERTER *my_converter = NULL;
55     LP_STATUS status;
56
57     if (converter == NULL)
58         return LP_STS_INVARG;
59
60     status = NC_algorithm_get_class_data(converter, ALGORITHM_CONVERTER,
61                                         (const void **)&my_converter);
62     if (LP_status_is_OK(status))
63         status = my_converter->converter(data, datal);
64     return status;
65 }

```

The parity algorithm (parity.c):

```

1  /* parity.c */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <LPmsg.h>
7  #include "converter.h"
8
9  static LP_STATUS set_even_parity(unsigned char *data, size_t datal)
10 {
11     for(; datal-- > 0; data++)
12     {
13         int bitcount = *data;
14
15         bitcount = ((bitcount & 0xAA) >> 1) + (bitcount & 0x55);
16         bitcount = ((bitcount & 0xCC) >> 2) + (bitcount & 0x33);
17         bitcount = ((bitcount & 0xF0) >> 4) + (bitcount & 0x0F);
18         if (bitcount & 1)
19             *data ^= 0x80;
20     }
21
22     return LP_STS_OK;
23 }
24
25 NC_ALGORITHM *create_even_parity_converter(void)
26 {

```

```

27  NC_ALGORITHM *converter = NULL;
28  LP_STATUS status =
29      create_converter("EvenParity", NULL, set_even_parity, &converter);
30
31  if (!LP_status_is_OK(status))
32      LP_signal_status(status, 0);
33  return converter;
34  }
35
36  int add_even_parity_converter(void)
37  {
38      NC_ALGORITHM *converter = create_even_parity_converter();
39      LP_STATUS status = NC_algorithm_register(converter);
40
41      if (!LP_status_is_OK(status))
42          {
43              LP_signal_status(status, 0);
44              return 0;
45          }
46
47      status = NC_algorithm_release_noconst(&converter);
48      if (!LP_status_is_OK(status))
49          {
50              LP_signal_status(status, 0);
51              return 0;
52          }
53
54      return 1;
55  }
56
57  int main()
58  {
59      const NC_ALGORITHM *converter = NULL;
60      unsigned char data[] = "This is a string to convert";
61      LP_STATUS status;
62
63      NC_init();
64
65      if (!add_even_parity_converter())
66          exit(1);
67
68      status = converter_algorithm("EvenParity", &converter);
69      if (!LP_status_is_OK(status))
70          {
71              LP_signal_status(status, 0);
72              exit(2);
73          }
74
75      printf("Before: %s\n", data);
76      convert(converter, data, strlen(data));
77      printf("After:  %s\n", data);
78
79      NC_finish();
80
81      exit(0);
82  }

```

The output from running parity should be:

```

1  Before: This is a string to convert
2  After:  Ôèiô ió á ótrifç to colðert

```