Tiny Crypto Library
Version 00.03

Tom St Denis
Algonquin College

tomstdenis@yahoo.com
http://tomstdenis.home.dhs.org
111 Banning Rd
Kanata, Ontario
K2L 1C3
Canada

December 21, 2001

Contents

Chapter 1

Introduction

1.1 What is libcrypt.a

Since I really could care less for a name you can call this library either "Tiny Crypto Library" or "libcrypt.a". I think there is already a "libcrypt.a" with "cygwin" but I could care less about that too.

So what is "libcrypt.a" or just "lba" for short? It is a portable ANSI C cryptographic library that supports symmetric ciphers, one-way hashes, pseudo-random number generators and public key crypto (via RSA). It is designed to compile out of the box with the GNU C Compiler (GCC) version 2.95.3 and higher.

The library is designed so new ciphers/hashes/prngs can be added and the existing API (and helper API functions) will be able to use the new designs automatically. There exist self-check functions for each cipher and hash to ensure that they compile and execute to the design specifications. The library also performs extensive parameter error checking and will give verbose error messages.

1.2 License

All of the code except for "mpi.c", "mpi-config.h" and "mpi.h" has been written from scratch by Tom St Denis and as such is under the official binding "Tom Doesn't Care About Licenses" or TDCAL license. This entitles the user to use this library for *whatever* they want without any form of repayment to the author whatsoever. Similarly MPI is public domain software copyright'ed by Michael Fromberger and may be used.

Essentially, "have fun kids."

1.3 Building the library

To build the library on a GCC equipped platform simply type “make” at your command prompt. It will build the library file “libcrypt.a”. If you are on a non-x86 platform comment out the appropriate line in “makefile”. To build the test executable just take off the “.exe”. I hope to have a makefile for *NIX platforms shortly.

To build the library on a non-GCC platform just build all of the source files (except test.c) and archive them in a single library.

To install the library copy all of the “.h” files into your include path and the single libcrypt.a file into your LIB path.

Et voila.

Chapter 2

The API

2.1 Introduction

In general the API is very simple to memorize and use. Most of the functions return either **void** or **int**. Functions that return **int** will return either **CRYPT_OK** or **CRYPT_ERROR**. If there is an error the character pointer **crypt_error**[1] will be set.

There is no initialization routine for the library and for the most part the code is thread safe. The only thread related issue is if you use the same symmetric cipher, hash or public key state data in multiple threads. Normally that is not an issue, therefore, the library is thread safe.

To include the prototypes for libcrypt into your own program simply include "crypt.h" like so:

```
#include <crypt.h>
int main(void) {
    return 0;
}
```

2.2 Macros

There are a few helper macros to make the coding process a bit easier. The first set are related to loading and storing 32/64-bit words in little endian format. The macros are:

STORE32L(x, y)	**unsigned long** x, **unsigned char** *y	$x \to y[0 \dots 3]$
STORE64L(x, y)	**unsigned long long** x, **unsigned char** *y	$x \to y[0 \dots 7]$
LOAD32L(x, y)	**unsigned long** x, **unsigned char** *y	$y[0 \dots 3] \to x$
LOAD64L(x, y)	**unsigned long long** x, **unsigned char** *y	$y[0 \dots 7] \to x$
BSWAP(x)	**unsigned long** x	Swaps the byte order of x.

[1]Note that it is lower case.

There are 32-bit cyclic rotations as well:

ROL(x, y)	**unsigned long** x, **unsigned long** y	$x << y$
ROR(x, y)	**unsigned long** x, **unsigned long** y	$x >> y$

2.3 Functions with Variable Length Output

Certain functions such as "rsa_export()" give an output that is variable length. To prevent buffer overflows you must pass it the length of the buffer[2] where the output will be stored. For example:

```
#include <crypt.h>
int main(void) {
    struct rsa_key key;
    unsigned char buffer[1024];
    int x;

    /* ... Make up the key somehow */

    /* lets export the key */
    x = 1024;
    if (rsa_export(buffer, &x, PK_PUBLIC, &key) == CRYPT_ERROR) {
       printf("Export error: %s\n", crypt_error);
       return -1;
    }

    /* ... do something with the buffer */

    return 0;
}
```

[2]Extensive error checking is not in place but it will be in future releases so it is a good idea to follow through with these guidelines.

Chapter 3

Symmetric Block Ciphers

3.1 Core Functions

libcrypt provides several block ciphers all in a plain vanila ECB block mode. All ciphers store their scheduled keys in a single union called “symmetric_key”. This allows all ciphers to have the same prototype and store their keys as naturally as possible. All ciphers provide four visible functions which are (given that XXX is the name of the cipher):

```
int XXX_setup(unsigned char *key, int keylen, int rounds,
              union symmetric_key *skey);
```

The XXX_setup() routine will setup the cipher to be used with a given number of rounds and a given key length (in bytes). The number of rounds can be set to zero to use the default. It returns **CRYPT_ERROR** if the parameters are invalid such as an incorrect key size or number of rounds. If the cipher is setup correctly it returns **CRYPT_OK**.

```
void XXX_ecb_encrypt(unsigned char *pt, unsigned char *ct,
                     union symmetric_key *skey);

void XXX_ecb_decrypt(unsigned char *ct, unsigned char *pt,
                     union symmetric_key *skey);
```

These two functions will encrypt or decrypt (respectively) a single block of text[1] and store the result where you want it. It is possible that the input and output buffer are the same buffer.

```
int XXX_test(void);
```

This function will return **CRYPT_OK** if the cipher matches the test vectors from the designs. It returns **CRYPT_ERROR** if it fails to meet the test vectors.

[1]The size of which depends on which cipher you are using.

An example snippet that encodes a block with Blowfish in ECB mode is below.

```
#include <crypt.h>
int main(void)
{
    unsigned char pt[8], ct[8], key[8];
    union symmetric_key skey;
    int x;

    /* make up a key */
    for (x = 0; x < 8; x++) pt[x] = key[x] = x;

    /* schedule the key */
    if (blowfish_setup(key, 8, 0, &skey) == CRYPT_ERROR) {
       printf("Setup error: %s\n", crypt_error);
       return -1;
    }

    /* encrypt the block */
    blowfish_ecb_encrypt(pt, ct, &skey);

    /* decrypt the block */
    blowfish_ecb_decrypt(ct, pt, &skey);

    return 0;
}
```

3.2 The Cipher Descriptors

To facilitate automatic routines an array of cipher descriptors is provided in the array "cipher_descriptor". An element of this array has the following format:

```
struct _cipher_descriptor {
   char *name;
   int  min_key_length, max_key_length,
        block_length, default_rounds;
   int  (*setup)      (unsigned char *key, int keylength,
                       int num_rounds, union symmetric_key *skey);
   void (*ecb_encrypt)(unsigned char *pt, unsigned char *ct,
                       union symmetric_key *key);
   void (*ecb_decrypt)(unsigned char *ct, unsigned char *pt,
                       union symmetric_key *key);
   int (*test)        (void);
};
```

Where “name” is the lower case ASCII version of the name. The fields “min_key_length”, “max_key_length” and “block_length” are all the number of bytes not bits. As a good rule of thumb it is assumed that the cipher supports the min and max key lengths but not always everything in between. The “default_rounds” field is the default number of rounds that will be used.

The remaining fields are all pointers to the core functions for each cipher. The end of the cipher_descriptor array is marked when “name” equals **NULL**.

As of this release the current cipher_descriptors elements are

Name	Block Size	Key Range	Rounds
Blowfish, “blowfish”	8	8 ... 56	16
RC5-32/12/b, “rc5”	8	8 ... 128	12 ... 24
RC6-32/20/b, “rc6”	16	8 ... 128	20
SAFER+, “safer+”	16	16, 24, 32	8, 12, 16
Serpent, “serpent”	16	16 .. 32	32

Note that “SAFER+” does not allow the user to override the number of rounds. The number of rounds varies per key length. There are 8 rounds for 16 byte keys, 12 rounds for 24 byte keys and 16 rounds for 32 byte keys. You can either specify the correct number of rounds or zero and let it pick.

To work with the cipher_descriptor array there is a function “int find_cipher(**char** *name)” that will search for a given name in the array. It returns negative one if the cipher is not found, otherwise it returns the location in the array where the cipher was found. For example, to indirectly setup Blowfish you can also use:

```
cipher_descriptor[find_cipher("blowfish")].setup(key, keylen, rounds, &skey);
```

A good safety would be to check the return value before accessing the desired function.

3.3 CBC and CTR Block Modes

The library provides simple support routines for handling CBC and CTR encoded messages. Assuming the mode you want is XXX there is a structure called “**struct symmetric_XXX**” that will contain the information required to use that mode. They have virtually identical setup routines:

```
int cbc_start(int cipher, unsigned char *IV, unsigned char *key,
              int keylen, int num_rounds, struct symmetric_CBC *cbc);

int ctr_start(int cipher, unsigned char *count, unsigned char *key,
              int keylen, int num_rounds, struct symmetric_CTR *ctr);
```

In both cases “cipher” is the index into the cipher_descriptor array of the cipher you want to use. The “IV” and “count” values are the initialization vectors. You must fill them yourself and it is assumed they are the same length as the block size of the cipher you choose. The parameters “key”, “keylen” and “num_rounds” are the same as in the XXX_setup() function call. The final parameter is a pointer to the structure you want to hold the information for the mode of operation.

Both routines return **CRYPT_OK** if the cipher initialized correctly, otherwise they return **CRYPT_ERROR**. To actually encrypt or decrypt the following routines are provided:

```
void cbc_encrypt(unsigned char *pt, unsigned char *ct,
                 struct symmetric_CBC *cbc);
void cbc_decrypt(unsigned char *ct, unsigned char *pt,
                 struct symmetric_CBC *cbc);

void ctr_encrypt(unsigned char *pt, unsigned char *ct,
                 int len, struct symmetric_CTR *ctr);
void ctr_decrypt(unsigned char *ct, unsigned char *pt,
                 int len, struct symmetric_CTR *ctr);
```

These routines work much like the ECB mode routines. In the CBC case it always encrypts blocks of a size determined by the cipher. This means in CBC mode you should pad your message (with zeroes will work) so the message length is a multiple of block cipher length. In CTR mode you can specify the size of each block which means when you reach the end of the message you can simply pass the value of the remaining length.

To decrypt in either mode you simply setup the CBC or CTR buffer like before (recall you have to fetch the IV value you used) and use the decrypt routine on all of the blocks.

Chapter 4

One-Way Hash Functions

4.1 Core Functions

Like the ciphers there are hash core functions and a universal structure to hold the hash state called "**union hash_state**". To initialize hash XXX (where XXX is the name) call:

```
void XXX_init(union hash_state *md);
```

This simply sets up the hash to the default state governed by the specifications of the hash. To add data to the message being hashed call:

```
void XXX_process(unsigned char *in, int len,
                 union hash_state *md);
```

Essentially all hash messages are virtually infinitely[1] long message which is buffered. You pass pieces of the data at a time to the process function. To finally get the result of the hash call:

```
void XXX_done(union hash_state *md,
              unsigned char *out);
```

This function will finish up the hash and store the result in the "out" array. You must ensure that "out" is long enough for the hash in question. To test a hash function call:

```
int XXX_test(void);
```

This will return **CRYPTO_OK** if the hash matches the test vectors, otherwise it returns **CRYPTO_ERROR**. An example snippet that hashes a message with md5 is given below.

[1]All hashes are limited to 2^{32} bits

```
#include <crypt.h>
int main(void)
{
    union hash_state md;
    unsigned char *in = "hello world", out[16];

    /* setup the hash */
    md5_init(&md);

    /* add the message */
    md5_process(in, strlen(in), &md);

    /* get the hash */
    md5_done(&md, out);

    return 0;
}
```

4.2 Hash Descriptors

Like the set of ciphers the set of hashes have descriptors too. They are stored in an array called "hash_descriptor" and are defined by:

```
struct _hash_descriptor {
    char *name;
    int hashsize;
    void (*init)   (union hash_state *);
    void (*process)(union hash_state *,
                    unsigned char *, int);
    void (*done)   (union hash_state *,
                    unsigned char *);
    int  (*test)   (void);
};
```

Similarly "name" is the name of the hash function in ASCII (all lowercase). "hashsize" is the size of the digest output in bytes. The remaining fields are pointers to the functions that do the respective tasks. There is a function to search the array as well called "int find_hash(char *name)". It returns -1 if the hash is not found, otherwise the position in the descriptor table of the hash.

There are two helper functions as well:

```
int hash_memory(int hash, unsigned char *data,
                int len, unsigned char *dst);

int hash_file(int hash, char *fname,
              unsigned char *dst);
```

Both functions return **CRYPT_OK** on success, otherwise they return **CRYPT_ERROR**. The "hash" parameter is the location in the descriptor table of the hash. The functions are otherwise straightforward. To perform the above hash with md5 the following code could be used:

```
hash_memory(find_hash("md5"), "hello world", 11, out);
```

The following hashes are provided as of this release:

Name	Size
SHA-256, "sha256"	32
SHA-1, "sha1"	20
MD5, "md5"	16
TIGER-192, "tiger"	24

Chapter 5

Pseudo-Random Number Generators

5.1 Core Functions

The library provides an array of core functions for PRNGs as well. There is a universal structure called "**union prng_state**". To initialize a PRNG call:

```
int XXX_start(union prng_state *prng);
```

This will setup the PRNG but not seed it. Returns **CRYPTO_ERROR** if there is an error. In order for the PRNG to be cryptographically useful you must give it entropy. Ideally you'd have some OS level source to tap like in UNIX. Since this is a portable library I leave it upto you to figure this one out. Nyah Nyah!. To add entropy to the PRNG call:

```
int XXX_add_entropy(unsigned char *in, int len,
                    union prng_state *prng);
```

Which returns **CRYPTO_OK** if the entropy was accepted. Once you think you have enough entropy you call another function to put the entropy into action.

```
int XXX_ready(union prng_state *prng);
```

Which returns **CRYPTO_OK** if it is ready. Finally to actually read bytes call:

```
int XXX_read(unsigned char *out, int len,
             union prng_state *prng);
```

Which returns the number of bytes read from the PRNG.

5.1.1 Remarks

It is possible to be adding entropy and reading from a PRNG at the same time. For example, if you first seed the PRNG and call ready() you can now read from it. You can also keep adding new entropy to it. The new entropy will not be used in the PRNG until ready() is called. This allows the PRNG to be used and re-seeded at the same time.

No real error checking is guaranteed to see if the entropy is sufficient or if the PRNG is even in a ready state. Simple snippet to read 10 bytes from yarrow is below.

```
#include <crypt.h>
int main(void)
{
   union prng_state prng;
   unsigned char buf[10];

   /* start it */
   yarrow_start(&prng);

   /* add entropy */
   yarrow_add_entropy("hello world", 11, &prng);

   /* ready and read */
   yarrow_ready(&prng);
   yarrow_read(buf, 10, &prng);

   return 0;
}
```

5.2 PRNG Descriptors

PRNGs have descriptors too (surprised?). Stored in the structure "prng_descriptor". The format of an element is:

```
struct _prng_descriptor {
    char *name;
    int (*start)      (union prng_state *);
    int (*add_entropy)(unsigned char *, int,
                       union prng_state *);
    int (*ready)      (union prng_state *);
    int (*read)       (unsigned char *, int len,
                       union prng_state *);
};
```

There is a "int find_prng(char *name)" function as well. Returns -1 if the PRNG is not found, otherwise it returns the position in the prng_descriptor array.

There are no additional support routines for PRNGs. Yarrow is the only currently support PRNG available. It uses by default SHA1 and Blowfish but you can change that after calling “yarrow_start()”.

Chapter 6

RSA Routines

6.1 Core Functions

For RSA routines a single "struct rsa_key" structure is used. To make a new RSA key call:

```
int rsa_make_key(union prng_state *prng,
                 int wprng, int size,
                 long e, struct rsa_key *key);
```

Where "wprng" is the index into the PRNG descriptor array. "size" is the size in bytes of the RSA modulus desired. "e" is the encryption exponent desired, typical values are 3, 17, 257 and 65537. I suggest you stick with 65537 since its big enough to prevent trivial math attacks and not super slow. "key" is where the key is placed. This routine returns **CRYPTO_ERROR** if it fails to make a RSA key.

Todo raw work with the RSA function call:

```
int rsa_exptmod(unsigned char *in, int inlen,
                unsigned char *out, int *outlen,
                int which, struct rsa_key *key);
```

This loads the bignum from "in" as a little endian word, raises it to either "e" or "d" and stores the result in "out" and the size of the result in "outlen". You must ensure that "outlen" is set to the maximum size of the output buffer before calling this function. This is to prevent buffer overruns. "which" is set to **PK_PUBLIC** to use "e" (i.e. for encryption) and set to **PK_PRIVATE** to use "d" as the exponent. This function returns **CRYPT_ERROR** on error.

6.2 Packet Routines

The remaining RSA functions are non-standard but should (to the best of my knowledge) be secure if used correctly. To encrypt a buffer of memory in a hybrid fashion call:

```
int rsa_encrypt(unsigned char *in, int len,
                unsigned char *out, int *outlen,
                union prng_state *prng, int wprng,
                int cipher, struct rsa_key *key);
```

This will encrypt the message with the cipher specified by “cipher” under a random key made by a PRNG specified by “wprng” and RSA encrypt the symmetric key with “key”. This stores all the relevent information in “out” and sets the length in “outlen”. You must ensure that “outlen” is set to the buffer size before calling this. This returns **CRYPT_ERROR** on error. To decrypt packets made by this routine call:

```
int rsa_decrypt(unsigned char *in, int len,
                unsigned char *out, int *outlen,
                struct rsa_key *key);
```

Which works akin to rsa_encrypt(). Similarly to sign/verify there are:

```
int rsa_sign(unsigned char *in, int inlen,
             unsigned char *out, int *outlen, int hash,
             union prng_state *prng, int wprng, struct rsa_key *key);

int rsa_verify(unsigned char *sig,
               unsigned char *msg,
               int inlen, int *stat,
               struct rsa_key *key);
```

The verify function sets “stat” to 1 if it passes or to 0 if it fails. The “sig” parameter is the output of the rsa_sign() function and “msg” is the original msg that was signed. To import/export RSA keys as a memory buffer (e.g. to store them to disk) call:

```
int rsa_export(unsigned char *out, int *outlen,
               int type, struct rsa_key *key);

int rsa_import(unsigned char *in, struct rsa_key *key);
```

To free the memory used by an RSA key call:

```
void rsa_free(struct rsa_key *key);
```