http://www.yassl.com
mailto:info@yassl.com
phone: +1 206 369 4800

CTaoCrypt Usage Reference

CTaoCrypt is the cryptography library primarily used by CyaSSL. It is optimized for speed, small footprint, and portability. CyaSSL can also interchange with other cryptography libraries as required.

Types used in the examples:

```
typedef unsigned char byte;
typedef unsigned int  word32;
```

Hash Functions

MD5

To use MD5 include the MD5 header "md5.h". The structure to use is Md5 which is a typedef. Before using the hash initialization must be done with the ***InitMd5()*** call. Use ***Md5Update()*** to update the hash and ***Md5Final()*** to retrieve the final hash:

```
byte md5sum[MD5_DIGEST_SIZE];
byte buffer[1024];
// fill buffer with data to hash

Md5 md5;
InitMd5(&md5);

Md5Update(&md5, buffer, sizeof(buffer));  // can be called again
and again
Md5Final(&md5, md5sum);
```

md5sum now contains the digest of the hashed data in buffer.

SHA

To use SHA include the SHA header "sha.h". The structure to use is Sha which is a typedef. Before using the hash initialization must be done with the ***InitSha()*** call. Use ***ShaUpdate()*** to update the hash and ***ShaFinal()*** to retrieve the final hash:

```
byte shaSum[SHA_DIGEST_SIZE];
byte buffer[1024];
// fill buffer with data to hash

Sha sha;
InitSha(&sha);

ShaUpdate(&sha, buffer, sizeof(buffer));    // can be called
again and again
ShaFinal(&sha, shaSum);
```

shaSum now contains the digest of the hashed data in buffer.

Other Hashes

Likewise, the same procedures can be used with MD4 "m4.h" (which is outdated and considered broken) and SHA-256 "sha256.h".

Message Digests

CTaoCrypt currently provides HMAC for message digest needs. The structure Hmac is found in the header "hmac.h". With HMAC initialization is done with ***HmacSetKey()***. 3 different types are supported with HMAC; MD5, SHA, and SHA-256. Here's an example with SHA-256.

```
Hmac    hmac;
byte    key[24];        // fill key with keying material
byte    buferr[2048];   // fill buffer with data to digest
byte    hmacDigest[SHA256_DIGEST_SIZE];

HmacSetKey(&hmac, SHA256, key, sizeof(key));
HmacUpdate(&hmac, buffer, sizeof(buffer));
HmacFinal(&hmac, hmacDigest);
```

hmacDigest now contains the digest of the hashed data in buffer.

Block Ciphers

DES and 3DES

CTaoCrypt provides support for DES and 3DES (Des3 since 3 is an invalid leading C identifier). To use these include the header "des.h". The structures you can use are Des and Des3. Initialization is done through ***Des_SetKey()*** or ***Des3_SetKey()***. CBC encryption/decryption is provided through ***Des_CbcEnrypt()*** / ***Des_CbcDecrypt()*** and ***Des3_CbcEncrypt()*** / ***Des_CbcDecrypt()***. Des has a key size of 8 bytes (24 for 3DES) and the block size is 8 bytes, so only pass increments of 8 bytes to encrypt/decrypt functions. In your data isn't in a block size increment you'll need to add padding to make sure it is. Each SetKey() also takes an IV, an initialization vector that is the same size as the key size. Usage is usually like the following:

```
Des3 enc;
Des3 dec;

const byte key[] = {  // some 24 byte key };
const byte iv[] = { // some 24 byte iv };

byte plain[24];   // an increment of 8, fill with data
byte cipher[24];

// encrypt
Des3_SetKey(&enc, key, iv, DES_ENCRYPTION);
Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

AES

CTaoCrypt also provides support for AES. Key sizes are 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits). CBC mode is supported for encrypt/decrypt. Please include the header "aes.h" to use AES. AES has a block size of 16 bytes and the IV should also be 16 bytes. The functions are exactly the same as DES and usage usually goes:

```
Aes enc;
Aes dec;

const byte key[] = {  // some 24 byte key };
const byte iv[] = { // some 16 byte iv };

byte plain[32];   // an increment of 16, fill with data
byte cipher[32];

// encrypt
Aes_SetKey(&enc, key, sizeof(key), iv, AES_ENCRYPTION);
Aes_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
Aes_SetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
Aes_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

Stream Ciphers

ARC4

The most common stream cipher used on the internet is ARC4 and CTaoCrypt supports it through the header "arc.h". Usage is simpler than block ciphers because there is no block size and the key length can be any length. Use it like this:

```
Arc4 enc;
Arc4 dec;

const byte key[] = {  // some key any length};

byte plain[27];   // no size restriction, fill with data
byte cipher[27];

// encrypt
Arc4SetKey(&enc, key, sizeof(key));
Arc4Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
Arc4SetKey(&dec, key, sizeof(key));
Arc4Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

RABBIT

A newer stream cipher gaining popularity is RABBIT and you can use it with CTaoCrypt by including the header "rabbit.h". RABBIT is very fast compared to ARC4 but has key constraints of 16 bytes (128 bits) and an optional IV of 8 bytes (64 bits). Otherwise usage is exactly like ARC4:

```
Rabbit enc;
Rabbit dec;

const byte key[] = {  // some key 16 bytes};
const byte iv[] = { // some iv 8 bytes };

byte plain[27];   // no size restriction, fill with data
byte cipher[27];

// encrypt
RabbitSetKey(&enc, key, iv);      // iv can be a NULL pointer
RabbitProcess(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
RabbitSetKey(&dec, key, iv);
RabbitProcess(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

HC-128

Another new stream cipher in current use is HC-128 which is even faster than RABBIT. To use it with CTaoCrypt please include the header "hc128.h". HC-128 also uses 16 bytes keys (128 bits) but uses 16 bytes vs (128 bits) unlike RABBIT.

```
HC128 enc;
HC128 dec;

const byte key[] = {  // some key 16 bytes};
const byte iv[] = { // some iv 16 bytes };

byte plain[37];   // no size restriction, fill with data
byte cipher[37];

// encrypt
Hc128_SetKey(&enc, key, iv);      // iv can be a NULL pointer
Hc128_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
Hc128_SetKey(&dec, key, iv);
Hc128_Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

Public Key Cryptography

RSA

CTaoCrypt provides support for RSA through the header "rsa.h". There are two types of RSA keys, public and private. A public key allows anyone to encrypt something that only the holder of the private key can decrypt. It also allows the private key holder to sign something and anyone with a public key can verify that only the private key holder actually signed it. Usage is usually like the following:

```
RsaKey rsaPublicKey;

byte publicKeyBuffer[]  = { // holds  the raw data from the key,
maybe from a file like RsaPublicKey.der };
word32 idx = 0;                   //  where to start reading
into the buffer

RsaPublicKeyDecode(publicKeyBuffer, &idx, &rsaPublicKey,
sizeof(publicKeyBuffer));

byte in[] = { // plain text to encrypt };
byte out[128];
RNG rng;

InitRng(&rng);

word32 outLen = RsaPublicEncrypt(in, sizeof(in), out,
sizeof(out), &rsaPublicKey, &rng);
```

Now out holds the cipher text from the plain text in. ***RsaPublicEncrypt()*** will return the length in bytes written to out or a negative number in case of an error. ***RsaPublicEncrypt()*** needs an RNG (Random Number Generator) for the padding used by the encryptor and it must be initialized before it can be used. To make sure that the output buffer is large enough to pass you can first call ***RsaEncryptSize()*** which will return the number of bytes that a successful call to ***RsaPublicEnrypt()*** will write.

In the event of an error, a negative return from ***RsaPublicEnrypt()***, or ***RsaPublicKeyDecode()*** for that matter, you can call ***CTaoCryptErrorString()*** to get a string describing the error that occurred.

```
void CTaoCryptErrorString(int error, char* buffer);
```

Make sure that buffer is at least **MAX_ERROR_SZ** bytes (80).

Now to decrypt out:

```
RsaKey rsaPrivateKey;
```

```
byte privateKeyBuffer[] = { // hold the raw data from the key,
maybe from a file like RsaPrivateKey.der };
word32 idx = 0;                   //  where to start reading
into the buffer

RsaPrivateKeyDecode(privateKeyBuffer, &idx, &rsaPrivateKey,
sizeof(privateKeyBuffer));

byte plain[128];

word32 plainSz = RsaPrivateDecrypt(out, outLen, plain,
                          sizeof(plain), &rsaPrivateKey);
```

Now plain will hold plainSz bytes or an error code.

For complete examples of each type in CTaoCrypt please see the file ctaocrypt/test.c.