yaSSL API Reference

09.20.2010

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

I. Overview of the TaoCrypt API

All TaoCrypt Hash functions are derived from the HASH Class which provides two very useful virtual functions:

```
virtual void Update(const byte*, word32) = 0;
virtual void Final(byte*) = 0;
```

Most Hashing needs can be fulfilled using these two functions, e.g.,

```
byte md5sum[MD5::DIGEST_SIZE];
byte buffer[1024];
// fill buffer
MD5 hash;
hash.Update(buffer, 1024);
// continue updating if needed
hash.Final(md5sum);
```

md5sum now contains the digest of the hashed data. The same code can be used for MD2, MD5, SHA-1, or RIPEMD. Using the base class can abstract away the differences:

```
void GetDigest(byte* digest, HASH& hash, const byte* input, word32 sz) {
        hash.Update(input, sz);
        hash.Final(digest);
}
```

Please see test.cpp for complete examples and test vectors.

A. Message Digests

TaoCrypt currently provides HMAC for message digest needs. Update() and Final() are provided, just like the hashing functions, but an additional function is needed for keying:

```
void SetKey(const byte*, word32);
```

HMAC is as easy to use as the hashes, just provide a hashing function and a key:

```
byte digest[SHA::DIGEST_SIZE];
byte buffer[1024];
byte key[16];
// fill buffer and key

HMAC<SHA> hmac;
hmac.SetKey(key, 16);
hmac.Update(buffer, 1024);
hmac.Final(digest);
```

B. Block Ciphers

TaoCrypt provides DES, 3DES, and AES for block cipher needs. Currently, ECB and CBC modes are supported (only what SSL and TLS require). Some typedefs are provided in the headers for these ciphers:

```
typedef BlockCipher<ENCRYPTION, DES, ECB> DES_ECB_Encryption;
```

The Block Ciphers rely on one function for encrypting and decrypting:

```
void Process(byte*, const byte*, word32);
```

Keying is simple:

```
void SetKey(const byte*, word32);
void SetKey(const byte*, word32, const byte*);
```

The second version is used when an initialization vector is needed. Usage is trivial, to encrypt:

```
byte key[8] = { // some key };
byte buffer[24] = { // some plain text };
byte cipherText[24];
DES_ECB_Encryption enc;
enc.SetKey(key, 8);
enc.Process(cipherText, buffer, 24);
```

Decrypting is simple as well:

```
byte plainText[24];
DES_ECB_Decryption dec;
dec.SetKey(key, 8);
dec.Process(plainText, cipherText, 24);
```

3DES and AES are used the same way, please see test.cpp for complete examples.

C. Stream Ciphers

TaoCrypt provides ARC4 for stream cipher needs. Like BlockCipher, ARC4 can be used with just two functions:

```
void SetKey(const byte*, word32);
void Process(byte*, const byte*, word32);
```

Typdefs are also provided in the header for encryption and decryption needs. Here is a simple example that encrypts:

```
byte key[16] = { // some key };
byte buffer[100] = { // some plain text };
byte cipherText[100];
ARC4::Encryption enc;
enc.SetKey(key, 16);
enc.Process(cipherText, buffer, 100);
```

Decrypting is analogous:

```
byte plainText[100];
ARC4::Decryption dec;
dec.SetKey(key, 16);
dec.Process(plainText, cipherText, 100);
```

Please see test.cpp for a complete example and test vectors.

D. Public Key Cryptography

TaoCrypt provides RSA and DSA for Public Key Cryptography.

RSA

There are public and private RSA keys, RSA_PublicKey and RSA_PrivateKey. Typically, they are initialized with a Source object, which is usually constructed with a byte array or a file:

```
byte privKey[64] = { // some key };
Source privSrc(privKey, 64);
RSA_PrivateKey rsaPriv(privSrc);
```

A public key can also be constructed from a private key:

```
RSA_PublicKey rsaPub(rsaPriv);
// or a file source example

Source fSrc;
FileSource("./rsaPublic.dat", fSrc);

RSA_PublicKey rsaPub(fSrc);
```

Once you have a public key, you can perform public encryption:

```
byte buffer[64] = { // plain Text };
byte cipher[64];
RandomNumberGenerator rng;
```

```
RSAES_Encryptor enc(rsaPub);
enc.Encrypt(buffer, 64, cipher, rng);
```

RSAES_Encryptor is a typedef for:

```
typedef RSA_Encryptor<> RSAES_Encryptor;
```

Where the default template argument is RSA_BlockType2, a padding scheme. Decrypting requires a private key:

```
byte plain[64];
RSAES_Decryptor dec(rsaPriv);
dec.Decrypt(cipher, 64, plain, rng);
```

TaoCrypt can also do RSA sign and verify operations, including the SSL type which require RSA_BlockType1 padding. The member functions SSL_Sign() and SSL_Verify() handle these and are used with the same arguments as encryption and decryption. TaoCrypt handles inversing the keys transparently. Please see test.cpp for a complete example.

DSA

There are public and private DSA keys, DSA_PublicKey and DSA_PrivateKey. Typically, they are initialized with a Source object, which is usually constructed with a byte array or a file:

```
byte privKey[128] = { // some key };
Source privSrc(privKey, 128);
DSA_PrivateKey dsaPriv(privSrc);
```

Once you have a private DSA key, you can sign a message, usually a SHA digest:

```
RandomNumberGenerator rng;
byte message[SHA::DIGEST_SIZE] = { // a hash };
byte signature[40];
DSA_Signer signer(dsaPriv);
signer.Sign(message, signature, rng);
```

Verifying is a simple operation as well:

```
DSA_Verifier verifier(dsaPriv); // or Public Key
bool result = verifier.Verify(message, signautre);
```

Please see test.cpp for a complete example.

Key Agreement

TaoCrypt provides Diffie-Hellman (DH) for key agreement arrangements. The DH class is generally constructed from a Source object to initialize p and g.

```
byte key[80] = { // contains p and g };
Source keySrc(key, 80);
DH self(keySrc);
```

Once you have a DH object, you generate a key pair:

```
byte pub[128];
byte priv[128];
RandomNumberGenerator rng;
self.GenerateKeyPair(rng, priv, pub);
```

Then you can send someone your public key and obtain their public one. Using your private key and their public key you can generate an agreement:

```
byte agree[128];
self.Agree(agree, priv, otherPublicKey);
```

The other person will get the same agreement. Please see test.cpp for a complete example.

II. Overview of the yaSSL API

A. Client Side

The yaSSL Client side API is declared in namespace yaSSL and is centered on the Client class. A "hello yaSSL" example will prove useful in showing the most basic usage.

```
yaSSL::Client client;
client.SetCa("ca-cert.pem");
yaSSL::SOCKET_T socket;
// ... do a tcp connect on socket here
client.Connect(socket);

client.Write("hello yaSSL", 12);
char buffer[80];
client.Read(buffer, 80);
std::cout << buffer << std::endl;
```

The Client class is declared in yassl.hpp, which doesn't include any headers itself. No arguments are needed for the constructor, and the only option that is usually set is for Certificate Authority files. These are Certificates that yaSSL will use for verification purposes. After that, the client object only needs a connected socket to perform an SSL connect. yaSSL uses the SOCKET_T type which is used for the differences between BSD style sockets and Windows ones. Please see test.hpp which includes tcp_connect(), all the examples use this function to perform TCP connects. The socket is the only parameter Connect() takes, its prototype is:

```
int Connect(SOCKET_T s);
```

Error handling has been omitted for clarity. Once an SSL connection has been established, transferring data is simple. Two functions handle the needs:

```
int Write(const void*, int);
int Read(void*, int);
```

Once you are done using the Client connection, simply let the object go out of scope (or delete it if it's on the heap), the destructor takes care of closing the connection and cleaning up any memory use. It's that easy.

B. Server Side

The server side yaSSL API is also in namespace yaSSL and is based on the Server class, declared in yassl.hpp. All of the same functions as the Client are supported, and the use is exactly the same. Two other functions are needed to control the Server's certificates:

```
void SetCert(const char*);
void SetKey(const char*);
```

Please see the examples in the download for further details.