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

CyaSSL Extensions Reference

1) Startup and Exit

All applications should call *InitCyaSSL()* before using the library and call *FreeCyaSSL()* at program termination. Currently these functions only initialize and free the shared mutex for the session cache in multi-user mode but in the future they may do more so it's always a good idea to use them.

2) Compression

CyaSSL supports data compression with the zlib library. The ./configure build system detects the presence of this library, if you're building in some other way define the constant **HAVE_LIBZ** and include the path to zlib.h for your includes. Compression is off by default for a given cipher, to turn it on, use the function *CyaSSL_set_compression()* before SSL connecting or accepting. Both the client and server must have compression turned on in order for compression to be used.

3) CyaSSL Debugging

CyaSSL has support for debugging through log messages in environments where debugging is limited. To turn logging on use the function *CyaSSL_Debugging_ON()* and to turn it off use *CyaSSL_Deubgging_OFF()*. In a normal build (release mode) these functions will have no effect. In a debug build define **DEBUG_CYASSL** to ensure these functions are turned on.

4) Domain Name check for server certificate

CyaSSL has an extension on the client that automatically checks the domain of the server certificate. In OpenSSL mode nearly a dozen function calls are needed to perform this. CyaSSL checks that the date of the certificate is in range, verifies the signature, and additionally verifies the domain if you call

```
 CyaSSL_check_domain_name(SSL* ssl, cons char* dn)
```

before calling *SSL_connect()*. CyaSSL will match the X509 issuer name of peer's server certificate against **dn** (the expected domain name). If the names match *SSL_connect()* will proceed normally, however if there is a name mismatch, *SSL_connect()* will return a fatal error and *SSL_get_error()* will return **DOMAIN_NAME_MISMATCH**.

Checking the domain name of the certificate is an important step that verifies the server is actually who it claims to be. This extension is intended to ease the burden of performing the check.

5) No Filesystem and using Certificates

Normally a filesystem is used to load private keys, certificates, and CAs. Since CyaSSL is sometimes used in environments without a full filesystem an extension to use memory buffers instead is provided. To use the extension define the constant **NO_FILESYSTEM** and the following functions will be made available:

```
int CyaSSL_CTX_load_verify_buffer(SSL_CTX*, const unsigned char*, long)
int CyaSSL_CTX_use_certificate_buffer(SSL_CTX*,const unsigned char*,long,int)
int CyaSSL_CTX_use_PrivateKey_buffer(SSL_CTX*,const unsigned char*,long,int)
int CyaSSL_CTX_use_certificate_chain_buffer(SSL_CTX*,
                                            const unsigned char*,long)
```

Use these functions exactly like their counterparts that are named *file* instead of *buffer*. And instead of providing a file name provide a memory buffer.

6) HandShake CallBack

CyaSSL has an extension that allows a HandShake CallBack to be set for connect or accept. Use the extended functions:

```
int CyaSSL_connect_ex(SSL*, HandShakeCallBack, TimeoutCallBack, Timeval)
int CyaSSL_accept_ex(SSL*, HandShakeCallBack, TimeoutCallBack, Timeval)
```

HandShakeCallBack is defined as:

```
 typedef int (*HandShakeCallBack)(HandShakeInfo*);
```

HandShakeInfo is defined in openssl/cyassl_callbacks.h (which should be added to a non-standard build):

```
  typedef struct handShakeInfo_st {
      char   cipherName[MAX_CIPHERNAME_SZ + 1];   /* negotiated cipher   */
      char   packetNames[MAX_PACKETS_HANDSHAKE][MAX_PACKETNAME_SZ+1];
                                              /* SSL packet names     */
      int    numberPackets;                   /* actual # of packets  */
      int    negotiationError;                /* cipher/parameter err */
  } HandShakeInfo;
```

No dynamic memory is used since the maximum number of SSL packets in a handshake exchange is known. Packet names can be accessed through *packetNames[idx]* up to *numberPackets*. The callback will be called whether or not a handshake error occured. Example usage is also in the client example.

7) Timeout Callback

The same extensions as above are used, they can call be called with either, both, or neither   callbacks. *TimeoutCallback* is defined as:

```
typedef int (*TimeoutCallBack)(TimeoutInfo*);
```

Where *TimeoutInfo* looks like:

```
typedef struct timeoutInfo_st {
   char       timeoutName[MAX_TIMEOUT_NAME_SZ + 1];  /* timeout Name */
   int        flags;                             /* for future use*/
   int        numberPackets;                     /* actual # of packets */
   PacketInfo packets[MAX_PACKETS_HANDSHAKE]; /* list of all packets  */
   Timeval    timeoutValue;                      /* timer that caused it */
 } TimeoutInfo;
```

Again, no dynamic memory is used for this structure since a maximum number of SSL packets is known for a handshake. *Timeval* is just a typedef for struct timeval.

PacketInfo is defined like this:

```
typedef struct packetInfo_st {
   char        packetName[MAX_PACKETNAME_SZ + 1]; /* SSL name */
   Timeval     timestamp;                /* when it occured    */
   unsigned char value[MAX_VALUE_SZ];    /* if fits, it's here */
   unsigned char* bufferValue;           /* otherwise here (non 0) */
   int         valueSz;                  /* sz of value or buffer */
} PacketInfo;
```

Here, dynamic memory may be used. If the SSL packet can fit in *value* then that's where it's placed. *valueSz* holds the length and *bufferValue* is 0. If the packet is too big for *value*, only **Certificate** packets should cause this, then the packet is placed in *bufferValue*. *valueSz* still holds the size.

If memory is allocated for a **Certificate** packet then it is reclaimed after the callback returns. The timeout is implemented using signals, specifically SIGALRM, and is thread safe. If a previous alarm is set of type ITIMER_REAL then it is reset, along with the correct handler, afterwards. The old timer will be time adjusted for any time CyaSSL spends processing. If an existing timer is shorter than the passed timer, the existing timer value is used. It is still reset afterwards. An existing timer that expires will be reset if has an interval associated with it. The callback will only be issued if a timeout occurs.

See the client example for usage.

8) Pre Shared Keys

CyaSSL has added support for two ciphers with pre shared keys:

TLS_PSK_WITH_AES_256_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA

These new suites are automatically built into CyaSSL though they can be turned off at build time with the constant **NO_PSK**. To only use these ciphers at runtime use the function *SSL_CTX_set_cipher_list()*.

On the client use the function *SSL_CTX_set_psk_client_callback()* to setup the callback. The client example in CyaSSL_Home/examples/client/client.c gives example usage for setting up the client identity and key, though the actual callback is implemented in exampes/test.h.

CyaSSL supports identities and hints up to 128 octets and pre shared keys up to 64 octets.

9) TLS 1.1 and 1.2

CyaSSL easily supports TLS 1.1 and TLS 1.2. You can use them by using the functions:

```
TLSv1_1_server_method(void);
TLSv1_1_client_method(void);
```

for TLS 1.1 or for TLS 1.2:

```
TLSv1_2_server_method(void);
TLSv1_2_client_method(void);
```

10) RSA Key Generation

CyaSSL supports RSA key generation of varying lengths up to 4096 bits. Key generation is off by default but can be turned on during the ./configure process with:

--enable-keygen

or by defining CYASSL_KEY_GEN in Windows or non-standard environments. Creating a key is easy, only requiring one function from rsa.h:

```
int MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);
```

Where *size* is the length in bits and *e* is the public exponent, using 65537 is usually a good choice for *e*. The following from ctaocrypt/test/test.c gives an example creating an RSA key of 1024 bits:

```
RsaKey genKey;
RNG    rng;
int    ret;

InitRng(&rng);
InitRsaKey(&genKey, 0);

ret = MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret < 0)
    /* ret contains error */;
```

The RsaKey *genKey* can now be used like any other RsaKey. If you need to export the key CyaSSL provides both DER and PEM formatting in asn.h. Always convert the key to DER format first, and then if you need PEM use the generic *DerToPem()* function like this:

```
byte der[4096];
int  derSz = RsaKeyToDer(&genKey, der, sizeof(der));
if (derSz < 0)
    /* derSz contains error */;
```

The buffer *der* now holds a DER format of the key. To convert the DER buffer to PEM use the conversion function:

```
byte pem[4096];
int  pemSz = DerToPem(der, derSz, pem, sizeof(pem),
                      PRIVATEKEY_TYPE);
```

```
if (pemSz < 0)
    /* pemSz contains error */;
```

The last argument of *DerToPem()* takes a type parameter, usually either *PRIVATEKEY_TYPE* or *CERT_TYPE*. Now the buffer *pem* holds the PEM format of the key.

11) Certificate Generation

CyaSSL now supports self-signed x509 v3 certificate generation. Certificate generation is off by default but can be turned on during the ./configure process with:

 --enable-certgen

or by defining CYASSL_CERT_GEN in Windows or non-stanard environments.

Before a certificate can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from asn.h named Cert:

```
/* for user to fill for certificate generation */
typedef struct Cert {
    int      version;                 /* x509 version  */
    byte     serial[SERIAL_SIZE];     /* serial number */
    int      sigType;                 /* signature algo type */
    CertName issuer;                  /* issuer info */
    int      daysValid;               /* validity days */
    int      selfSigned;              /* self signed flag */
    CertName subject;                 /* subject info */
} Cert;
```

Where CertName looks like:

```
typedef struct CertName {
    char country[NAME_SIZE];
    char state[NAME_SIZE];
    char locality[NAME_SIZE];
    char org[NAME_SIZE];
    char unit[NAME_SIZE];
    char commonName[NAME_SIZE];
    char email[NAME_SIZE];
} CertName;
```

Before filling in the subject information an initialization function needs to be called like this:

```
Cert myCert;
InitCert(&myCert);
```

InitCert() sets defaults for some of the variables including setting the *version* to 3 (0x02), the *serial* number to 0 (randomly generated), the *sigType* to MD5_WITH_RSA, the *daysValid* to 500, and *selfSigned* to 1 (TRUE). Currently only MD5_WITH_RSA (by far the most common) and self signed are supported though the next release will allow other signers and other signature types.

Now the user can initialize the subject information like this example from ctaocrypt/test/test.c

```
strncpy(myCert.subject.country, "US", NAME_SIZE);
strncpy(myCert.subject.state, "OR", NAME_SIZE);
strncpy(myCert.subject.locality, "Portland", NAME_SIZE);
strncpy(myCert.subject.org, "yaSSL", NAME_SIZE);
strncpy(myCert.subject.unit, "Development", NAME_SIZE);
strncpy(myCert.subject.commonName, "www.yassl.com", NAME_SIZE);
strncpy(myCert.subject.email, "info@yassl.com", NAME_SIZE);
```

Then the certificate can be generated using the variables *genKey* and *rng* from the above key generation example (of course any valid RsaKey or RNG can be used):

```
byte derCert[4096];

int certSz = MakeCert(&myCert, derCert, sizeof(derCert), &key,
                      &rng);
if (certSz < 0)
    /* certSz contains the error */;
```

The buffer *derCert* now contains a DER format of the certificate. If you need a PEM format of the certificate you can use the generic DerToPem function and specify the type to be *CERT_TYPE* like this:

```
byte pemCert[4096];

int pemCertSz = DerToPem(derCert, certSz, pemCert,
                         sizeof(pemCert), CERT_TYPE);
if (pemCertSz < 0)
    /* pemCertSz contains error */;
```

Now the buffer *pemCert* holds the PEM format of the certificate.