



<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.

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 occurred. 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 occurred */
    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 `examples/test.h`.

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

Copyright (C) 2010 Sawtooth Consulting Ltd. All rights reserved.