wolfSSL User Manual

April, 2019
Version 4.0.0

Table of Contents

Chapter 1: Introduction

This manual is written as a technical guide to the wolfSSL embedded SSL/TLS library. It will explain how to build and get started with wolfSSL, provide an overview of build options, features, portability enhancements, support, and much more.

Why Choose wolfSSL?

There are many reasons to choose wolfSSL as your embedded SSL solution. Some of the top reasons include size (typical footprint sizes range from 20-100 kB), support for the newest standards (SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3, DTLS 1.0, and DTLS 1.2), current and progressive cipher support (including stream ciphers), multi-platform, royalty free, and an OpenSSL compatibility API to ease porting into existing applications which have previously used the OpenSSL package. For a complete feature list, see **Section 4.1**.

Chapter 2: Building wolfSSL

wolfSSL was written with portability in mind and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please don't hesitate to seek support through our **support forums** (http://www.wolfssl.com/forums) or contact us directly at **support@wolfssl.com**.

This chapter explains how to build wolfSSL on Unix and Windows, and provides guidance for building wolfSSL in a non-standard environment. You will find the "getting started" guide in **Chapter 3** and an SSL tutorial in **Chapter 11**.

When using the autoconf / automake system to build wolfSSL, wolfSSL uses a single Makefile to build all parts and examples of the library, which is both simpler and faster than using Makefiles recursively.

2.1 Getting wolfSSL Source Code

The most recent version of wolfSSL can be downloaded from the wolfSSL website as a ZIP file:

http://wolfssl.com/wolfSSL/download/downloadForm.php

After downloading the ZIP file, unzip the file using the `unzip`"unzip" command. To use native line endings, enable the "-a" modifier when using unzip. From the unzip man page, the "-a" modifier functionality is described:

> [...] The `-a` option causes files identified by `zip` as text files (those with the '`t`' label in `zipinfo` listings, rather than '`b`') to be automatically extracted as such, converting line endings, end-of-file characters and the character set itself as necessary. [...]

NOTE: Beginning with the release of wolfSSL 2.0.0rc3, the directory structure of wolfSSL was changed as well as the standard install location. These changes were made to make it easier for open source projects to integrate wolfSSL. For more information on header and structure changes, please see **Sections 9.1** and **9.3**.

2.2 Building on *nix

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other *nix-like systems, use the autoconf system. To build wolfSSL you only need to run two commands from the wolfSSL root directory:

```
./configure
make
```

You can append any number of build options to `./configure`. For a list of available build options, please see **Section 2.5** or run:

```
./configure --help
```

from the command line to see a list of possible options to pass to the `./configure` script. To build wolfSSL, run:

```
make
```

To install wolfSSL run:

```
make install
```

You may need superuser privileges to install, in which case precede the command with sudo:

```
sudo make install
```

To test the build, run the `testsuite` program from the root wolfSSL directory:

```
./testsuite/testsuite.test
```

or use autoconf to run the testsuite as well as the standard wolfSSL API and crypto tests:

```
make test
```

Further details about expected output of the `testsuite` program can be found in **Section 3.2**. If you want to build only the wolfSSL library and not the additional items (examples, testsuite, benchmark app, etc.), you can run the following command from the wolfSSL root directory:

```
make src/libwolfssl.la
```

2.3 Building on Windows

In addition to the instructions below, you can find instructions and tips for building wolfSSL with Visual Studio here.

VS 2008: Solutions are included for Visual Studio 2008 in the root directory of the install. For use with Visual Studio 2010 and later, the existing project files should be able to be converted during the import process.

Note:
If importing to a newer version of VS you will be asked: “Do you want to overwrite the project and its imported property sheets?” You can avoid the following by selecting “No”. Otherwise if you select “Yes”, you will see warnings about EDITANDCONTINUE being ignored due to SAFESEH specification. You will need to right click on the testsuite, sslSniffer, server, echoserver, echoclient, and client individually and modify their Properties->Configuration Properties->Linker->Advanced (scroll all the way to the bottom in Advanced window). Locate “Image Has Safe Exception Handlers” and click the drop down arrow on the far right. Change this to No (/SAFESEH:NO) for each of the aforementioned. The other option is to disable EDITANDCONTINUE which, we have found to be useful for debugging purposes and is therefore not recommended.

VS 2010: You will need to download Service Pack 1 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean and rebuild the project; the linker error should be taken care of.

VS 2013 (64 bit solution): You will need to download Service Pack 4 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean the project then Rebuild the project and the linker error should be taken care of.

To test each build, choose “Build All” from the Visual Studio menu and then run the testsuite program. To edit build options in the Visual Studio project, select your desired project (wolfssl, echoclient, echoserver, etc.) and browse to the “Properties” panel.

Note:

After the wolfSSL v3.8.0 release the build preprocessor macros were moved to a centralized file located at 'IDE/WIN/user_settings.h'. This file can also be found in the project. To add features such as ECC or ChaCha20/Poly1305 add #defines here such as HAVE_ECC or HAVE_CHACHA / HAVE_POLY1305.

Cygwin: If using Cygwin, or other toolsets for Windows that provides *nix-like commands and functionality, please follow the instructions in section 2.2, above, for "Building on *nix". If building wolfSSL for Windows on a Windows development machine, we recommend using the included Visual Studio project files to build wolfSSL.

2.4 Building in a non-standard environment

While not officially supported, we try to help users wishing to build wolfSSL in a non-standard environment, particularly with embedded and cross-compilation systems. Below are some notes on getting started with this.

1. The source and header files need to remain in the same directory structure as they are in the wolfSSL download package.

2. Some build systems will want to explicitly know where the wolfSSL header files are located, so you may need to specify that. They are located in the <wolfssl_root>/wolfssl directory. Typically, you can add the <wolfssl_root> directory to your include path to resolve header problems.

3. wolfSSL defaults to a little endian system unless the configure process detects big endian. Since users building in a non-standard environment aren't using the configure process, `BIG_ENDIAN_ORDER` will need to be defined if using a big endian system.

4. wolfSSL benefits speed-wise from having a 64-bit type available. The configure process determines if `long` or `long long` is 64 bits and if so sets up a define. So if `sizeof(long)` is 8 bytes on your system, define `SIZEOF_LONG 8`. If it isn't but `sizeof(long long)` is 8 bytes, then define `SIZEOF_LONG_LONG 8`.

5. Try to build the library, and let us know if you run into any problems. If you need help, contact us at info@wolfssl.com.

6. Some defines that can modify the build are listed in the following subsections, below. For more verbose descriptions of many options, please see section 2.5.1,

"Build Option Notes".

2.4.1 Building into Yocto Linux

wolfSSL also includes recipes for building wolfSSL on Yocto Linux and OpenEmbedded. These recipes are maintained within the **meta-wolfSSL** layer as a GitHub repository, here: https://github.com/wolfSSL/meta-wolfssl. Building wolfSSL on Yocto Linux will require Git and bitbake. The following steps list how to get some wolfSSL products (that recipes exist for) built on Yocto Linux.

1. **Cloning wolfSSL meta**
   This can be done through a git-clone command of the following URL: https://github.com/wolfSSL/meta-wolfssl
2. **Insert the "meta-wolfSSL" layer into the build's bblayers.conf**
   Within the BBLAYERS section, add the path to the location where meta-wolfssl was cloned into. Example:
   ```
   BBLAYERS ?= "... \
   /path/to/meta-wolfssl/ \
   ..."
   ```
3. **Build a wolfSSL product recipe**
   bitbake can be used to build one of the three following wolfSSL product recipes: *wolfssl*, *wolfssh*, and *wolfmqtt*. Simply pass one of those recipes into the bitbake command (example: `bitbake wolfssl`). This allows the user to personally confirm compilation succeeds without issues.
4. **Edit local.conf**
   The final step is to edit the build's local.conf file, which allows desired libraries to be included with the image being built. Edit the "IMAGE_INSTALL_append" line to include the name of the desired recipe(s). An example of this is shown below:
   ```
   IMAGE_INSTALL_apped = "wolfssl wolfssh wolfmqtt"
   ```

Once the image has been built, wolfSSL's default location (or related products from recipes) will be the "`/usr/lib/`" directory.

Additionally, wolfSSL can be customized when building into Yocto by using the enable and disable options listed in section 2.5. This requires creating a .bbappend file and placing it within the wolfSSL application/recipe layer. The contents of this file should include a line specifying content to concatenate onto the `EXTRA_OECONF` variable. An example of this is shown below to enable TLS 1.3 support through the TLS 1.3 enable option:

```
EXTRA_OECONF += "--enable-tls13"
```

Further documentation on building into Yocto can be found in the **meta-wolfssl README**, located here: https://github.com/wolfSSL/meta-wolfssl/blob/master/README.md

2.4.2 Building with Atollic TrueSTUDIO

Versions of wolfSSL following 3.15.5 include a TrueSTUDIO project file that is used to build wolfSSL on ARM M4-Cortex devices. The TrueSTUDIO IDE simplifies the process of building on STM32 devices, is free to download, and is created by Atollic - a part of STMicroelectronics. To build the wolfSSL static library project file in TrueSTUDIO, it will require the user perform the following steps after opening TrueSTUDIO:

1. Import the project into the workspace (File > Import)
2. Build the project (Project > Build project)

The build then includes the settings located inside of **user_settings.h** at build-time. The default content of the **user_settings.h** file is minimal, and does not contain many features. Users are able to modify this file and add or remove features with options listed in the remainder of this chapter.

2.4.3 Building with Renesas CS+

Versions of wolfSSL following 3.15.0 include Renesas CS+ project files that can be used to build wolfSSL inside of the Renesas CS+ IDE. The CS+ IDE is compatible with other Renesas hardware tools. The steps for building wolfSSL through CS+ are located in one of the wolfSSL READMEs, **<wolfssl-root>/IDE/Renesas/CS+/Projects/README**.

2.4.4 Removing Features

The following defines can be used to remove features from wolfSSL. This can be helpful if you are trying to reduce the overall library footprint size. In addition to defining a `NO_<feature-name>` define, you can also remove the respective source file as well from the build (but not the header file).

`NO_WOLFSSL_CLIENT` removes calls specific to the client and is for a server-only builds. You should only use this if you want to remove a few calls for the sake of size.

NO_WOLFSSL_SERVER likewise removes calls specific to the server side.

NO_DES3 removes the use of DES3 encryptions. DES3 is built-in by default because some older servers still use it and it's required by SSL 3.0.

NO_DH and **NO_AES** are the same as the two above, they are widely used.

NO_DSA removes DSA since it's being phased out of popular use.

NO_ERROR_STRINGS disables error strings. Error strings are located in src/internal.c for wolfSSL or wolfcrypt/src/asn.c for wolfCrypt.

NO_HMAC removes HMAC from the build.

NO_MD4 removes MD4 from the build, MD4 is broken and shouldn't be used.

NO_MD5 removes MD5 from the build.

NO_SHA256 removes SHA-256 from the build.

NO_PSK turns off the use of the pre-shared key extension. It is built-in by default.

NO_PWDBASED disables password-based key derivation functions such as PBKDF1, PBKDF2, and PBKDF from PKCS #12.

NO_RC4 removes the use of the ARC4 stream cipher from the build. ARC4 is built-in by default because it is still popular and widely used.

NO_RABBIT and **NO_HC128** remove stream cipher extensions from the build.

NO_SESSION_CACHE can be defined when a session cache is not needed. This should reduce memory use by nearly 3 kB.

NO_TLS turns off TLS. We don’t recommend turning off TLS.

SMALL_SESSION_CACHE can be defined to limit the size of the SSL session cache used by wolfSSL. This will reduce the default session cache from 33 sessions to 6 sessions and save approximately 2.5 kB.

WC_NO_RSA_OAEP removes code for OAEP padding.

NO_AES_CBC turns off AES-CBC algorithm support.

NO_AES_128, **NO_AES_192**, and **NO_AES_256** can be defined to determine the selection of key sizes at compile-time.

NO_DEV_URANDOM disables the use of "/dev/urandom"

WOLFSSL_IGNORE_FILE_WARN disables the use of build warnings when not building with autotools

2.4.5 Enabling Features Disabled by Default

WOLFSSL_CERT_GEN turns on wolfSSL's certificate generation functionality. See **Chapter 7** for more information.

WOLFSSL_DER_LOAD allows loading DER-formatted CA certs into the wolfSSL context (`WOLFSSL_CTX`) using the function `wolfSSL_CTX_der_load_verify_locations()`.

WOLFSSL_DTLS turns on the use of DTLS, or datagram TLS. This isn't widely supported or used.

WOLFSSL_ENCRYPTED_KEYS turns on support for wolfSSL to use PEM encrypted keys using password callback without requiring "opensslextra".

WOLFSSL_KEY_GEN turns on wolfSSL's RSA key generation functionality. See **Chapter 7** for more information.

WOLFSSL_RIPEMD enables RIPEMD-160 support.

WOLFSSL_SHA384 enables SHA-384 support.

WOLFSSL_SHA512 enables SHA-512 support.

DEBUG_WOLFSSL builds in the ability to debug. For more information regarding debugging wolfSSL, see Chapter 8.

HAVE_AESCCM enables AES-CCM support.

HAVE_AESGCM enables AES-GCM support.

WOLFSSL_AES_XTS enables AES-XTS support.

HAVE_CAMELLIA enables Camellia support.

HAVE_CHACHA enables ChaCha20 support.

HAVE_POLY1305 enables Poly1305 support.

HAVE_CRL enables Certificate Revocation List (CRL) support.

HAVE_CRL_IO enables blocking inline HTTP request on the CRL URL. It will load the CRL into the WOLFSSL_CTX and apply it to all WOLFSSL objects created from it.

HAVE_ECC enables Elliptical Curve Cryptography (ECC) support.

HAVE_LIBZ is an extension that can allow for compression of data over the connection. It normally shouldn't be used, see the note below under configure notes libz.

HAVE_OCSP enables Online Certificate Status Protocol (OCSP) support.

OPENSSL_EXTRA builds even more OpenSSL compatibility into the library, and enables the wolfSSL OpenSSL compatibility layer to ease porting wolfSSL into existing applications which had been designed to work with OpenSSL. It is off by default.

TEST_IPV6 turns on testing of IPv6 in the test applications. wolfSSL proper is IP neutral, but the testing applications use IPv4 by default.

HAVE_CSHARP turns on configuration options needed for C# wrapper.

HAVE_CURVE25519 turns on the use of curve25519 algorithm.

HAVE_ED25519 turns on use of the ed25519 algorithm.

CURVED25519_SMALL defines CURVE25519_SMALL and ED25519_SMALL.

CURVE25519_SMALL use small memory option for curve25519. This uses less memory, but is slower.

ED25519_SMALL use small memory option for ed25519. This uses less memory, but is slower.

WOLFSSL_DH_CONST turns off use of floating point values when performing Diffie Hellman operations and uses tables for `XPOW()` and `XLOG()`. Removes dependency on external math library.

WOLFSSL_TRUST_PEER_CERT turns on the use of trusted peer certificates. This allows for loading in a peer certificate to match with a connection rather than using a CA. When turned on if a trusted peer certificate is matched than the peer cert chain is not loaded and the peer is considered verified. Using CAs is preferred.

WOLFSSL_STATIC_MEMORY turns on the use of static memory buffers and functions. This allows for using static memory instead of dynamic.

WOLFMEM_IO_SZ can be defined to specify the desired static IO buffer size.

WOLFSSL_SESSION_EXPORT turns on the use of DTLS session export and import. This allows for serializing and sending/receiving the current state of a DTLS session.

WOLFSSL_ARMASM turns on the use of ARMv8 hardware acceleration.

WC_RSA_NONBLOCK turns on fast math RSA non-blocking support for splitting RSA operations into smaller chunks of work. Feature is enabled by calling wc_RsaSetNonBlock and checking for FP_WOULDBLOCK return code.
WOLFSSL_RSA_VERIFY_ONLY turns on small build for RSA verify only use. Should be used with the macros WOLFSSL_RSA_PUBLIC_ONLY,

WOLFSSL_RSA_VERIFY_INLINE, NO_SIG_WRAPPER, and WOLFCRYPT_ONLY.

WOLFSSL_RSA_PUBLIC_ONLY turns on small build for RSA public key only use. Should be used with the macro WOLFCRYPT_ONLY.

WOLFSSL_RSA_PUBLIC_ONLY turns on small build for RSA public key only use. Should be used with the macro WOLFCRYPT_ONLY.

WOLFSSL_SHA3 turns on build for SHA3 use. This is support for SHA3 Keccak for the sizes SHA3-224, SHA3-256, SHA3-384 and SHA3-512. In addition WOLFSSL_SHA3_SMALL can be used to trade off performance for resource use.

2.4.6 Customizing or Porting wolfSSL

WOLFSSL_ALLOW_MAX_FRAGMENT_ADJUST if defined enables a feature which allows the maximum fragment size to be adjusted post-handshake.

WOLFSSL_MAX_SIGALGO provides the ability to override the maximum number of signature algorithms.

WOLFSSL_USER_SETTINGS if defined allows a user specific settings file to be used. The file must be named “user_settings.h” and exist in the include path. This is included prior to the standard “settings.h” file, so default settings can be overridden.

WOLFSSL_CALLBACKS is an extension that allows debugging callbacks through the use of signals in an environment without a debugger, it is off by default. It can also be used to set up a timer with blocking sockets. Please see **Chapter 6** for more information.

WOLFSSL_USER_IO allows the user to remove automatic setting of the default I/O functions `EmbedSend()` and `EmbedReceive()`. Used for custom I/O abstraction layer (see **Section 5.1** for more details).

NO_FILESYSTEM is used if stdio isn't available to load certificates and key files.

This enables the use of buffer extensions to be used instead of the file ones.

NO_INLINE disables the automatic inlining of small, heavily used functions. Turning this on will slow down wolfSSL and actually make it bigger since these are small functions, usually much smaller than function call setup/return. You'll also need to add wolfcrypt/src/misc.c to the list of compiled files if you're not using autoconf.

NO_DEV_RANDOM disables the use of the default /dev/random random number generator. If defined, the user needs to write an OS-specific GenerateSeed() function (found in "wolfcrypt/src/random.c").

NO_MAIN_DRIVER is used in the normal build environment to determine whether a test application is called on its own or through the testsuite driver application. You'll only need to use it with the test files: test.c, client.c, server.c, echoclient.c, echoserver.c, and testsuite.c

NO_WRITEV disables simulation of `writev()` semantics.

NO_MULTIBYTE_PRINT is used to compile out special characters that some embedded devices may have difficulty processing.

SINGLE_THREADED is a switch that turns off the use of mutexes. wolfSSL currently only uses one for the session cache. If your use of wolfSSL is always single threaded you can turn this on.

USER_TICKS allows the user to define their own clock tick function if `time(0)` is not wanted. Custom function needs second accuracy, but doesn't have to be correlated to Epoch. See `LowResTimer()` function in "wolfssl_int.c".

USER_TIME disables the use of time.h structures in the case that the user wants (or needs) to use their own. See "wolfcrypt/src/asn.c" for implementation details. The user will need to define and/or implement `XTIME()`, `XGMTIME()`, and `XVALIDATE_DATE()`.

USE_CERT_BUFFERS_1024 enables 1024-bit test certificate and key buffers located in <wolfssl_root>/wolfssl/certs_test.h. Helpful when testing on and porting to embedded systems with no filesystem.

USE_CERT_BUFFERS_2048 enables 2048-bit test certificate and key buffers located in <wolfssl_root>/wolfssl/certs_test.h. Helpful when testing on and porting to embedded systems with no filesystem.

USE_WOLF_TIME_T defining this option enables the library to use an internal time struct value.

USE_WOLF_STRTOK can be defined to allow the library to use the wolfSSL implementation of the function `strtok`.

USE_ECDSA_KEYSZ_HASH_ALGO can be defined to limit the size of the signature and algorithm hash digests to the ephemeral key sizes.

CUSTOM_RAND_GENERATE_SEED allows user to define custom function equivalent to `wc_GenerateSeed(byte* output, word32 sz).`

CUSTOM_RAND_GENERATE_BLOCK allows user to define custom random number generation function.
Examples of use are as follows.

```
./configure --disable-hashdrbg
CFLAGS="-DCUSTOM_RAND_GENERATE_BLOCK= custom_rand_generate_block"
```

or

```
/* RNG */
/* #define HAVE_HASHDRBG */
extern int custom_rand_generate_block(unsigned char* output,
                                      unsigned int sz);
```

WOLFSSL_SSLKEYLOGFILE allows the user to inspect the Pre-Master-Secret and other information when capturing packets with Wireshark. This option requires that **SHOW_SECRETS** be defined as well. These options are both HIGHLY INSECURE, and under no circumstances should these options be used within production builds.

WC_MAX_SYM_KEY_SIZE defines the max size of symmetric keys in the wolfSSL library. If not defined by the user, wolfSSL will define this macro as one eighth of max AES key size (if AES is enabled).

WOLFSSL_BASE16 allows the user to enable base 16 encoding or decoding.

2.4.7 Reducing Memory Usage

TFM_TIMING_RESISTANT can be defined when using fast math (USE_FAST_MATH) on systems with a small stack size. This will get rid of the large static arrays.

WOLFSSL_SMALL_STACK can be used for devices which have a small stack size. This increases the use of dynamic memory in wolfcrypt/src/integer.c, but can lead to slower performance.

RSA_LOW_MEM when defined CRT is not used which saves on some memory but slows down RSA operations. It is off by default.

USE_SLOW_SHA, USE_SLOW_SHA256, USE_SlOW_SHA512 when defined these macros reduce memory and resource usage for their corresponding SHA algorithms, at the cost of speed.

WOLFSSL_SHA3_SMALL when SHA3 is enabled this macro will reduce build size.

2.4.8 Increasing Performance

WOLFSSL_AESNI enables use of AES accelerated operations which are built into some Intel and AMD chipsets. When using this define, the aes_asm.c file must be added to the wolfSSL build sources.

USE_FAST_MATH switches the big integer library to a faster one that uses assembly if possible. fastmath will speed up public key operations like RSA, DH, and DSA. The big integer library is generally the most portable and generally easiest to get going with, but the negatives to the normal big integer library are that it is slower and it uses a lot of dynamic memory. Because the stack memory usage can be larger when using fastmath, we recommend defining TFM_TIMING_RESISTANT as well when using this option.

2.4.9 Stack or Chip Specific Defines

wolfSSL can be built for a variety of platforms and TCP/IP stacks. Most of the following defines are located in ./wolfssl/wolfcrypt/settings.h and are commented out by default.

Each can be uncommented to enable support for the specific chip or stack referenced below.

IPHONE can be defined if building for use with iOS.

THREADX can be defined when building for use with the ThreadX RTOS (www.rtos.com).

MICRIUM can be defined when building wolfSSL to enable support for Micrium's µC/OS-III RTOS (www.micrium.com).

MBED can be defined when building for the mbed prototyping platform (www.mbed.org).

MICROCHIP_PIC32 can be defined when building for Microchip's PIC32 platform (www.microchip.com).

MICROCHIP_TCPIP_V5 can be defined specifically version 5 of microchip tcp/ip stack.

MICROCHIP_TCPIP can be defined for microchip tcp/ip stack version 6 or later.

WOLFSSL_MICROCHIP_PIC32MZ can be defined for PIC32MZ hardware cryptography engine.

FREERTOS can be defined when building for FreeRTOS (www.freertos.org). If using LwIP, define `WOLFSSL_LWIP` as well.

FREERTOS_WINSIM can be defined when building for the FreeRTOS windows simulator (www.freertos.org).

EBSNET can be defined when using EBSnet products and RTIP.

WC_RNG_BLOCKING can be defined to enable block with sleep.

WOLFSSL_LWIP can be defined when using wolfSSL with the LwIP TCP/IP stack (http://savannah.nongnu.org/projects/lwip/).

WOLFSSL_GAME_BUILD can be defined when building wolfSSL for a game console.

WOLFSSL_LSR can be defined if building for LSR.

FREESCALE_MQX can be defined when building for Freescale MQX/RTCS/MFS (www.freescale.com). This in turn defines FREESCALE_K70_RNGA to enable support for the Kinetis H/W Random Number Generator Accelerator

WOLFSSL_STM32F1 can be defined when building for STM32F1.

WOLFSSL_STM32F2 can be defined when building for STM32F2. This define also enables STM32F2 hardware crypto and hardware RNG support in wolfSSL. (http://www.st.com/internet/mcu/subclass/1520.jsp)

COMVERGE can be defined if using Comverge settings.

WOLFSSL_QL can be defined if using QL SEP settings.

WOLFSSL_EROAD can be defined building for EROAD.

WOLFSSL_IAR_ARM can be defined if build for IAR EWARM.

WOLFSSL_TIRTOS can be defined when building for TI-RTOS.

WOLFSSL_ROWLEY_ARM can be defined when building with Rowley CrossWorks.

WOLFSSL_NRF51 can be defined when porting to Nordic nRF51.

WOLFSSL_NRF51_AES can be defined to use built-in AES hardware for AES 128 ECB encrypt when porting to Nordic nRF51.

WOLFSSL_CONTIKI can be defined to enable support for the Contiki operating system.

WOLFSSL_APACHE_MYNEWT can be defined to enable the Apache Mynewt port layer.

ASIO_USE_WOLFSSL can be defined to make wolfSSL build as an ASIO-compatible version. ASIO then relies on the **BOOST_ASIO_USE_WOLFSSL**

preprocessor define.

2.5 Build Options (`./configure` Options)

The following are options which may be appended to the `./configure` script to customize how the wolfSSL library is built.

By default, wolfSSL only builds in shared mode, with static mode being disabled. This speeds up build times by a factor of two. Either mode can be explicitly disabled or enabled if desired.

Option	Default Value	Description
`--enable-16bit`	Disabled	Enable 16 bit support
`--enable-32bit`	Disabled	Enable 32 bit support
`--enable-aes`	Enabled	Enable AES
`--enable-aescbc`	Enabled	Used to with --disable-aescbc to compile out AES-CBC
`--enable-aesccm`	Disabled	Enable AES-CCM support
`--enable-aescfb`	Disabled	Turns on AES-CFB mode support
`--enable-aesctr`	Disabled	Enable wolfSSL AES-CTR support
`--enable-aesgcm`	Enabled	Enable AES-GCM support
`--enable-aeskeywrap`	Disabled	Enable AES key wrap support
`--enable-aesni`	Disabled	Enable wolfSSL Intel AES-NI support
`--enable-afalg`	Disabled	Enables use of Linux module AF_ALG for hardware acceleration. Additional Xilinx use with =xilinx, =xilinx-rsa, =xilinx-aes, =xilinx-sha3
`--enable-all`	Disabled	Enables all wolfSSL features, excluding SSL v3
`--enable-alpn`	Disabled	Enable Application Layer Protocol Negotiation (ALPN)

--enable-anon	Disabled	Enable Anonymous
--enable-arc4	Disabled	Enable ARC4
--enable-asio	Disabled	Enables ASIO
--enable-asm	Enabled	Enable options for assembly, enables all inline assembly.
--enable-asn	Enabled	Enable ASN
--enable-asynccrypt	Disabled	Enable Asynchronous Crypto
--enable-atomicuser	Disabled	Enable Atomic User Record Layer
--enable-base16	Disabled	Base 16 encoding/decoding
--enable-base64encode	Enabled on x86_64	Enable Base64 encoding
--enable-bigcache	Disabled	Enable a big session cache
--enable-blake2	Disabled	Enable wolfSSL BLAKE2 support
--enable-bump	Disabled	Enable SSL Bump build
--enable-camellia	Disabled	Enable Camellia support
--enable-certgen	Disabled	Enable cert generation
--enable-certreq	Disabled	Enable cert request generation
--enable-certservice	Disabled	Enable certificate service (Windows Servers)
--enable-chacha	Enabled	Enable CHACHA
--enable-coding	Enabled	Enable Coding base 16/64
--enable-compkey	Disabled	Enable compressed keys support
--enable-crl	Disabled	Enable CRL
--enable-crl-monitor	Disabled	Enable CRL Monitor
--enable-cryptocb	Disabled	Enable crypto callbacks
--enable-cryptonly	Disabled	Enable wolfCrypt Only build
--enable-crypttests	Enabled	Enable Crypt Bench/Test

--enable-curve25519	Disabled	Enable Curve25519 (or `--enable-curve25519=small` for CURVE25519_SMALL)
--enable-debug	Disabled	Enable wolfSSL debugging support
--enable-des3	Enabled	Enable DES3
--enable-devcrypto	Disabled	Enables use of Linux /dev/crypto for hardware acceleration
--enable-dh	Enabled	Enable DH
--enable-distro	Disabled	Enable wolfSSL distro build
--enable-dsa	Disabled	Enable Digital Signature Algorithm (DSA)
--enable-dtls	Disabled	Enable wolfSSL DTLS support
--enable-ecc	Enabled on x86_64	Enable ECC
--enable-ecccustcurves	Disabled	Enable ECC custom curves
--enable-eccencrypt	Disabled	Enable ECC encrypt
--enable-eccshamir	Enabled on x86_64	Enable ECC Shamir
--enable-ed25519	Disabled	Enable ED25519 (or `--enable-ed25519=small` for ED25519_SMALL)
--enable-enckeys	Disabled	Enable PEM encrypted private key support
--enable-errorstrings	Enabled	Enable error strings table
--enable-examples	Enabled	Enable examples
--enable-extended-master	Enabled	Enable Extended Master Secret
--enable-fast-rsa	Disabled	Enable RSA using Intel IPP
--enable-fasthugemath	Disabled	Enable fast math + huge code
--enable-fastmath	Enabled on x86_64	Enable fast math ops

--enable-filesystem	Enabled	Enable Filesystem support
--enable-fips	Disabled	Enable FIPS 140-2 (Must have license to implement.)
--enable-fpecc	Disabled	Enable Fixed Point cache ECC
--enable-harden	Enabled	Enable Hardened build, Enables Timing Resistance and Blinding
--enable-hashdrbg	Enabled	Enable Hash DRBG support
--enable-hc128	Disabled	Enable streaming cipher HC-128
--enable-hkdf	Disabled	Enable HKDF (HMAC-KDF)
--enable-hugecache	Disabled	Enable a huge session cache
--enable-idea	Disabled	Enable IDEA Cipher
--enable-inline	Enabled	Enable inline functions
--enable-intelasm	Disabled	Enable ASM speedups for Intel and AMD processors
--enable-iopool	Disabled	Enable I/O Pool example
--enable-ipv6	Disabled	Enable testing of IPv6, wolfSSL proper is IP neutral
--enable-jni	Disabled	Enable wolfSSL JNI
--enable-jobserver [=no/ yes/#]	yes	Enable up to # make jobs yes: enable one more than CPU count
--enable-keygen	Disabled	Enable key generation
--enable-leanpsk	Disabled	Enable Lean PSK build
--enable-leantls	Disabled	Implements a lean TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.
--enable-lighty	Disabled	Enable lighttpd/lighty
--enable-lowresource	Disabled	Enable low resource usage options for memory/flash
--enable-maxfragment	Disabled	Enable Maximum Fragment Length

--enable-maxstrength	Disabled	Enable Max Strength build, allows TSLv1.2-AEAD-PFS ciphers only
--enable-mcapi	Disabled	Enable Microchip API
--enable-mcast	Disabled	Enable wolfSSL DTLS multicast support
--enable-md2	Disabled	Enable MD2 support
--enable-md4	Disabled	Enable MD4
--enable-md5	Enabled	Enable MD5
--enable-memory	Enabled	Enable memory callbacks
--enable-nginx	Disabled	Enable wolfSSL support for use with NGINX web servers
--enable-nullcipher	Disabled	Enable wolfSSL NULL cipher support (no encryption)
--enable-ocsp	Disabled	Enable Online Certificate Status Protocol (OCSP)
--enable-ocspstapling	Disabled	Enable OCSP Stapling
--enable-ocspstapling2	Disabled	Enable OCSP Stapling version 2
--enable-oldnames	Enabled	Enable backward-compatibility with old names
--enable-oldtls	Enabled	Enable old TLS version < 1.2
--enable-openssh	Disabled	Enable OpenSSH compatibility build
--enable-opensslall	Disabled	Enable all OpenSSL API and compatibility (large size increase)
--enable-opensslextra	Disabled	Enable some extra OpenSSL API compatibility (size increase)
--enable-optflags	Enabled	Enable default optimization CFLAGS for the compiler
--enable-pkcallbacks	Disabled	Enable Public Key Callbacks
--enable-pkcs11	Disabled	Enable PKCS#11 access
--enable-pkcs7	Disabled	Enable PKCS#7 support
--enable-poly1305	Enabled	Enable wolfSSL POLY1305 support

--enable-psk	Disabled	Enable PSK (Pre Shared Keys)
--enable-pwdbased	Disabled	Enable PWDBASED
--enable-rabbit	Disabled	Enable streaming cipher RABBIT
--enable-renegotiation-indication	Disabled	Enable Renegotiation Indication
--enable-ripemd	Disabled	Enable wolfSSL RIPEMD-160 support
--enable-rng	Enabled	Enable compiling and using RNG
--enable-rsa	Enabled	Enable RSA
--enable-rsapss	Disabled	Enable RSA-PSS
--enable-rsapub	Disabled	Enabled RSA public key only support (note requires --enable-cryptonly)
--enable-rsavfy	Disabled	Enables RSA verify only support (note requires --enable-cryptonly)
--enable-savecert	Disabled	Enable persistent cert cache
--enable-savesession	Disabled	Enable persistent session cache
--enable-scep	Disabled	Enable wolfSCEP (Simple Certificate Enrollment Protocol)
--enable-scrypt	Disabled	Enable SCRYPT
--enable-sctp	Disabled	Enable wolfSSL DTLS-SCTP support
--enable-secure-renegotiation	Disabled	Enable Secure Renegotiation
--enable-sep	Disabled	Enable SEP extensions
--enable-session-ticket	Disabled	Enable Session Ticket
--enable-sessioncerts	Disabled	Enable session cert storing
--enable-sessionexport	Disabled	Enable export and import of sessions
--enable-sha	Enabled	Enable SHA
--enable-sha224	Enabled on x86_64	Enable wolfSSL SHA-224 support

--enable-sha3	Disabled (except x86_64/ Aarch64)	Enables wolfSSL SHA3 support (=small for small build)
--enable-sha512	Enabled on x86_64	Enable wolfSSL SHA-512 support
--enable-shared[=PKGS]	Disabled	Building shared wolfSSL libraries [default = no]
--enable-singlethreaded	Disabled	Enable single threaded mode, no multi thread protections
--enable-smallcache	Disabled	Enable small session cache
--enable-smallstack	Disabled	Enable Small Stack Usage
--enable-sni	Disabled	Enable Server Name Indication (SNI)
--enable-sniffer	Disabled	Enable wolfSSL sniffer support
--enable-sp	Disabled	Enable single-precision math for RSA, DH, and ECC to improve performance.
--enable-sp-asm	Disabled	Enable single-precision assembly implementation
--enable-srp	Disabled	Enable Secure Remote Password
--enable-sslv3	Disabled	Enable SSL version 3.0
--enable-stacksize	Disabled	Enable stack size info on examples
--enable-static[=PKGS]	Disabled	Building static wolfSSL libraries [default=no]
--enable-staticmemory	Disabled	Enable static memory use
--enable-stunnel	Disabled	Enable stunnel
--enable-supportedcurves	Disabled	Enable Supported Elliptic Curves
--enable-testcert	Disabled	Enable Test Cert
--enable-tls13	Disabled	Enable TLS 1.3 support
--enable-tlsv12	Enabled	Enable TLS 1.2 support
--enable-tlsv10	Disabled	Enable TLS 1.0 support
--enable-tlsx	Disabled	Enable all TLS extensions

--enable-trackmemory	Disabled	Enable memory use information on wolfCrypt and wolfSSL cleanup
--enable-truncatedhmac	Disabled	Enable Truncated Keyed-hash MAC (HMAC)
--enable-valgrind	Disabled	Enable valgrind for unit tests
--enable-webclient	Disabled	Enable web client
--enable-webserver	Disabled	Enable Web Server
--enable-ssh	Disabled	Enable wolfSSH options
--enable-x963kdf	Disabled	Enable X9.63 KDF support
--enable-xilinx	Disabled	Enable wolfSSL support for Xilinx hardened encryption
--enable-xts	Disabled	Enables AES-XTS mode
--with-cavium=PATH	Disabled	Path to cavium/software directory.
--with-intelqa=PATH	Disabled	Path to Intel QuickAssist (QAT) driver directory.
--with-libz=PATH	Disabled	Optionally include libz for compression
--with-ntru=PATH	Disabled	Path to NTRU install (default /usr/)
--with-user-crypto=PATH	Disabled	Path to USER_CRYPTO install (default /usr/local).

2.5.1 Build Option Notes

tls13 - this build option can be combined with "--disable-tlsv12" and "--disable-oldtls" to produce a wolfSSL build that is only TLS 1.3.

debug - enabling debug support allows easier debugging by compiling with debug information and defining the constant **DEBUG_WOLFSSL** which outputs messages to **stderr**. To turn debug on at runtime, call *wolfSSL_Debugging_ON()*. To turn debug logging off at runtime, call *wolfSSL_Debugging_OFF()*. For more information, see **Chapter 8**.

singlethreaded - enabling single threaded mode turns off multi thread protection of the session cache. Only enable single threaded mode if you know your application is single threaded or your application is multithreaded and only one thread at a time will be accessing the library.

dtls - enabling DTLS support allows users of the library to also use the DTLS protocol in addition to TLS and SSL. For more information, see **Chapter 4**.

opensslextra - enabling OpenSSL Extra includes a larger set of OpenSSL compatibility functions. The basic build will enable enough functions for most TLS/SSL needs, but if you're porting an application that uses 10s or 100s of OpenSSL calls, enabling this will allow better support. The wolfSSL OpenSSL compatibility layer is under active development, so if there is a function missing which you need, please contact us and we'll try to help. For more information about the OpenSSL Compatibility Layer, please see **Chapter 13**.

ipv6 - enabling IPV6 changes the test applications to use IPv6 instead of IPv4. wolfSSL proper is IP neutral, either version can be used, but currently the test applications are IP dependent, IPv4 by default.

leanpsk - Very small build using PSK, and eliminating many features from the library. Approximate build size for wolfSSL on an embedded system with this enabled is 21kB.

fastmath - enabling fastmath will speed up public key operations like RSA, DH, and DSA. By default, wolfSSL uses the normal big integer math library. This is generally the most portable and generally easiest to get going with. The negatives to the normal big integer library are that it is slower and it uses a lot of dynamic memory. This option switches the big integer library to a faster one that uses assembly if possible. Assembly inclusion is dependent on compiler and processor combinations. Some combinations will need additional configure flags and some may not be possible. Help with optimizing fastmath with new assembly routines is available on a consulting basis.

On ia32, for example, all of the registers need to be available so high optimization and omitting the frame pointer needs to be taken care of. wolfSSL will add "-O3 -fomit-frame-pointer" to **GCC** for non debug builds. If you're using a different compiler you may need to add these manually to **CFLAGS** during configure.

OS X will also need "-mdynamic-no-pic" added to CFLAGS. In addition, if you're building in shared mode for ia32 on OS X you'll need to pass options to LDFLAGS as well:

```
LDFLAGS="-Wl,-read_only_relocs,warning"
```

This gives warning for some symbols instead of errors.

fastmath also changes the way dynamic and stack memory is used. The normal math library uses dynamic memory for big integers. Fastmath uses fixed size buffers that hold 4096 bit integers by default, allowing for 2048 bit by 2048 bit multiplications. If you need 4096 bit by 4096 bit multiplications then change **FP_MAX_BITS** in wolfssl/wolfcrypt/tfm.h. As FP_MAX_BITS is increased, this will also increase the runtime stack usage since the buffers used in the public key operations will now be larger. A couple of functions in the library use several temporary big integers, meaning the stack can get relatively large. This should only come into play on embedded systems or in threaded environments where the stack size is set to a low value. If stack corruption occurs with fastmath during public key operations in those environments, increase the stack size to accommodate the stack usage.

If you are enabling fastmath without using the autoconf system, you'll need to define USE_FAST_MATH and add tfm.c to the wolfSSL build instead of integer.c.

Since the stack memory can be large when using fastmath, we recommend defining TFM_TIMING_RESISTANT when using the fastmath library. This will get rid of large static arrays.

fasthugemath - enabling fasthugemath includes support for the fastmath library and greatly increases the code size by unrolling loops for popular key sizes during public key operations. Try using the benchmark utility before and after using fasthugemath to see if the slight speedup is worth the increased code size.

bigcache - enabling the big session cache will increase the session cache from 33 sessions to 20,027 sessions. The default session cache size of 33 is adequate for TLS clients and embedded servers. The big session cache is suitable for servers that aren't under heavy load, basically allowing 200 new sessions per minute or so.

hugecache - enabling the huge session cache will increase the session cache size to 65,791 sessions. This option is for servers that are under heavy load, over 13,000 new sessions per minute are possible or over 200 new sessions per second.

smallcache - enabling the small session cache will cause wolfSSL to only store 6 sessions. This may be useful for embedded clients or systems where the default of nearly 3kB is too much RAM. This define uses less than 500 bytes of RAM.

savesession - enabling this option will allow an application to persist (save) and restore the wolfSSL session cache to/from memory buffers.

savecert - enabling this option will allow an application to persist (save) and restore the wolfSSL certificate cache to/from memory buffers.

atomicuser - enabling this option will turn on User Atomic Record Layer Processing callbacks. This will allow the application to register its own MAC/encrypt and decrypt/verify callbacks.

pkcallbacks - enabling this option will turn on Public Key callbacks, allowing the application to register its own ECC sign/verify and RSA sign/verify and encrypt/decrypt callbacks.

sniffer - enabling sniffer (SSL inspection) support will allow the collection of SSL traffic packets as well as the ability to decrypt those packets with the correct key file. Currently the sniffer supports the following RSA ciphers

CBC ciphers:

- AES-CBC
- Camellia-CBC
- 3DES-CBC

Stream ciphers:

- RC4
- Rabbit
- HC-128

aesgcm - enabling AES-GCM will add these cipher suites to wolfSSL. wolfSSL offers four different implementations of AES-GCM balancing speed versus memory consumption. If available, wolfSSL will use 64-bit or 32-bit math. For embedded applications, there is a speedy 8-bit version that uses RAM-based lookup tables (8KB per session) which is speed comparable to the 64-bit version and a slower 8-bit version that doesn't take up any additional RAM. The --enable-aesgcm configure option may be modified with the options "=word32", "=table", or "=small", i.e. "--enable-aesgcm=table".

aesccm - enabling AES-GCM will enable Counter with CBC-MAC Mode with 8−byte authentication (CCM-8) for AES.

aesni - enabling AES-NI support will allow AES instructions to be called directly from the chip when using an AES-NI supported chip. This provides speed increases for AES functions. See **Chapter 4** for more details regarding AES-NI.

poly1305 - enabling this option will add Poly1305 support to wolfCrypt and wolfSSL.

camellia - enabling this option will add Camellia-CBC support to wolfCrypt and wolfSSL.

chacha - enabling this option will add ChaCha support to wolfCrypt and wolfSSL.

md2 - enabling this option adds support for the MD2 algorithm to wolfSSL. MD2 is disabled by default due to known security vulnerabilities.

ripemd - enabling this option adds support for the RIPEMD-160 algorithm to wolfSSL.

sha512 - enabling this option adds support for the SHA-512 hash algorithm. This algorithm needs the word64 type to be available, which is why it is disabled by default. Some embedded system may not have this type available.

sessioncerts - enabling this option adds support for the peer's certificate chain in the session cache through the wolfSSL_get_peer_chain(), wolfSSL_get_chain_count(), wolfSSL_get_chain_length(), wolfSSL_get_chain_cert(), wolfSSL_get_chain_cert_pem(), and wolfSSL_get_sessionID() functions.

keygen - enabling support for RSA key generation allows generating keys of varying lengths up to 4096 bits. wolfSSL provides both DER and PEM formatting.

certgen - enables support for self-signed X.509 v3 certificate generation.

certreq - enabling this option will add support for certificate request generation.

hc128 - Though we really like the speed of the HC-128 streaming cipher, it takes up some room in the cipher union for users who aren't using it. To keep the default build small in as many aspects as we can, we've disabled this cipher by default. In order to use this cipher or the corresponding cipher suite just turn it on, no other action is required.

rabbit - enabling this option adds support for the RABBIT stream cipher.

psk - Pre Shared Key support is off by default since it's not commonly used. To enable this feature simply turn it on, no other action is required.

poly1305 - enabling this option adds support for Poly1305 to wolfcrypt and wolfSSL.

webserver - this turns on functions required over the standard build that will allow full functionality for building with the yaSSL Embedded Web Server.

noFilesystem - this makes it easier to disable filesystem use. This option defines NO_FILESYSTEM.

inline - disabling this option disables function inlining in wolfSSL. Function placeholders that are not linked against but, rather, the code block is inserted into the function call when function inlining is enabled.

ecc - enabling this option will build ECC support and cipher suites into wolfSSL.

ocsp - enabling this option adds OCSP (Online Certificate Status Protocol) support to wolfSSL. It is used to obtain the revocation status of x.509 certificates as described in RFC 6960.

crl - enabling this option adds CRL (Certificate Revocation List) support to wolfSSL.

crl-monitor - enabling this option adds the ability to have wolfSSL actively monitor a specific CRL (Certificate Revocation List) directory.

ntru - this turns on the ability for wolfSSL to use NTRU cipher suites. NTRU is now available under the GPLv2 from Security Innovation. The NTRU bundle may be downloaded from the Security Innovation GitHub repository available at https://github.com/NTRUOpenSourceProject/ntru-crypto.

sni - enabling this option will turn on the TLS Server Name Indication (SNI) extension.

maxfragment - enabling this option will turn on the TLS Maximum Fragment Length extension.

truncatedhmac - enabling this option will turn on the TLS Truncated HMAC extension.

supportedcurves - enabling this option will turn on the TLS Supported ECC Curves extension.

tlsx - enabling this option will turn on all TLS extensions currently supported by wolfSSL.

valgrind - enabling this option will turn on valgrind when running the wolfSSL unit tests.

This can be useful for catching problems early on in the development cycle.

testcert - when this option is enabled, it exposes part of the ASN certificate API that is usually not exposed. This can be useful for testing purposes, as seen in the wolfCrypt test application (wolfcrypt/test/test.c).

examples - this option is enabled by default. When enabled, the wolfSSL example applications will be built (client, server, echoclient, echoserver).

gcc-hardening - enabling this option will add extra compiler security checks.

jobserver - enabling this option allows "make" on computers with multiple processors to build several files in parallel, which can significantly reduce build times. Users have the ability to pass different arguments to this command (yes/no/#). If "yes" is used, the configure script will tell make to use one more than the CPU count for the number of jobs. "no" obviously disables this feature. Optionally, the user can pass in the number of jobs as well.

disable shared - disabling the shared library build will exclude a wolfSSL shared library from being built. By default only a shared library is built in order to save time and space.

disable static - disabling the static library build will exclude a wolfSSL static library from being built. This options is enabled by default. A static library can be built by using the --enable-static build option.

libz - enabling libz will allow compression support in wolfSSL from the libz library. Think twice about including this option and using it by calling *wolfSSL_set_compression()* . While compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

fast-rsa - enabling fast-rsa speeds up RSA operations by using IPP libraries. It has a larger memory consumption then the default RSA set by wolfSSL. If IPP libraries can not be found an error message will be displayed during configuration. The first location that autoconf will look is in the directory wolfssl_root/IPP the second is standard location for libraries on the machine such as /usr/lib/ on linux systems.

The libraries used for RSA operations are in the directory "wolfssl-X.X.X/IPP/" where X.X.X is the current wolfSSL version number. Building from the bundled libraries is dependent on the directory location and name of IPP so the file structure of the subdirectory IPP should not be changed.

When allocating memory the fast-rsa operations have a memory tag of DYNAMIC_TYPE_USER_CRYPTO. This allows for viewing the memory consumption of RSA operations during run time with the fast-rsa option.

leantls - enabling produces a small footprint TLS client that supports TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

curve25519 - an elliptic curve offering 128 bits of security and to be used with ECDH key agreement (see § 2.6 Cross Compiling).

renegotiation-indication - as described in RFC 5746, this specification prevents an SSL/TLS attack involving renegotiation splicing by tying the renegotiations to the TLS connection they are performed over.

scep - as defined by the Internet Engineering Task Force, Simple Certificate Enrollment Protocol is a PKI that leverages PKCS#7 and PKCS#10 over HTTP. CERT notes that SCEP does not strongly authenticate certificate requests.

dsa - NIST approved digital signature algorithm along with RSA and ECDSA as defined by FIPS 186-4 and are used to generate and verify digital signatures if used in conjunction with an approved hash function as defined by the Secure Hash Standard (FIPS 180-4).

curve25519 - enabling curve25519 option allows for the use of the curve25519 algorithm. The default curve25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option -”-enable-curve25519=small” can be used. Although using less memory there is a trade off in speed.

ed25519 - enabling ed25519 option allows for the use of the ed25519 algorithm. The default ed25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option ”--enable-ed25519=small” can be used. Like with curve25519 using this enable option less is a trade off between speed and memory. As ed25519 is one of the algorithms now included in the base specification for TLS 1.3, wolfSSL also provides support for using ed25519 with TLS 1.3 and TLS 1.2.

Intelasm - enabling the intelasm option for wolfSSL will utilize expanded capabilities of your processor that dramatically enhance AES performance. The instruction sets leveraged when configure option is enabled include AVX1, AVX2, BMI2, RDRAND, RDSEED, AESNI, and ADX. These were first introduced into Intel processors and AMD processors have started adopting them in recent years. When enabled, wolfSSL will check the processor and take advantage of the instruction sets your processor supports.

sp-asm can be used to enable single-precision performance improvements through assembly with ARM and 64-bit ARM architectures.

armasm the default configure sets mcpu or mfpu based on 64 vs 32 bit system. It does not overwrite mcpu or mfpu setting passed in by use of CPPFLAGS. On some compilers -mstrict-align may be needed due to the m constraints and -mstrict-align is now also set by default unless a user passes in mcpu/mfpu flags with CPPFLAGS.

asio requires that the options "opensslextra" and "opensslall" be enabled when configuring wolfSSL. If these two options are not enabled, then the autoconf tool will automatically enable these options to enable ASIO when configuring wolfSSL.

devcrypto has the ability to receive arguments, being able to receive any combination of *aes* (all aes support), *hash* (all hash algorithms), and *cbc* (aes-cbc only). If no options are given, it will default to using *all* (*aes, hash*).

cryptodev allows wolfSSL users to utilize the Linux kernel's built in cryptographic drivers through encryption callbacks. Note that is feature is distinctly different from devcrypto, and since versions 4.0.0 has been deprecated in favor of the cryptocb option.

cryptocb allows users to utilize the Linux kernel's built in cryptographic drivers though encryption callbacks.

afalg is similar to devcrypto in that it leverages a Linux kernel module (AF_ALG) for offloading crypto operations. On some hardware the module has performance accelerations available through the Linux crypto drivers. In the case of Petalinux with Xilinx the flag --enable-afalg=xilinx can be used to tell wolfSSL to use the Xilinx interface for AF_ALG.

nginx is used to enable wolfSSL support for use with NGINX and NGINX web servers. This option can be combined with "tls13" or other configure options that specify TLS versions.

optflags can be disabled to turn off the default flags and settings that are defined in the settings.h file used by wolfSSL, allowing the user to define their own flags.

2.6 Cross Compiling

Many users on embedded platforms cross compile wolfSSL for their environment. The easiest way to cross compile the library is to use the ./configure system. It will generate a Makefile which can then be used to build wolfSSL. However, more recent versions (post 3.12.2) include an example Makefile for GCC-ARM architectures. This Makefile can be found in **<wolfssl-root>/IDE/GCC-ARM/Makefile**.

When cross compiling, you'll need to specify the host to ./configure, such as:

```
./configure --host=arm-linux
```

You may also need to specify the compiler, linker, etc. that you want to use:

```
./configure --host=arm-linux CC=arm-linux-gcc AR=arm-linux-ar
RANLIB=arm-linux
```

The C language standard can also be changed by either defining the macro `WOLF_C99` in the user's settings file, or by defining it within CFLAGS when running the configure script.

There is a bug in the configure system which you might see when cross compiling and detecting user overriding malloc. If you get an undefined reference to 'rpl_malloc' and/or 'rpl_realloc', please add the following to your ./configure line:

```
ac_cv_func_malloc_0_nonnull=yes ac_cv_func_realloc_0_nonnull=yes
```

After correctly configuring wolfSSL for cross-compilation, you should be able to follow standard autoconf practices for building and installing the library:

```
make
sudo make install
```

If you have any additional tips or feedback about cross compiling wolfSSL, please let us know at info@wolfssl.com.

2.6.1 Example cross compile configure options for toolchain builds

Toolchain	Description
`armebv7-eabihf-glibc`	./configure --host=armeb-linux \ CC=armeb-linux-gcc LD=armeb-linux-ld \ AR=armeb-linux-ar \ RANLIB=armeb-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
`armv5-eabi-glibc`	./configure --host=arm-linux \ CC=arm-linux-gcc LD=arm-linux-ld \ AR=arm-linux-ar \ RANLIB=arm-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
`armv6-eabihf-glibc`	./configure --host=arm-linux \ CC=arm-linux-gcc LD=arm-linux-ld \ AR=arm-linux-ar \ RANLIB=arm-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
`armv7-eabihf-glibc`	./configure --host=arm-linux \ CC=arm-linux-gcc LD=arm-linux-ld \ AR=arm-linux-ar \ RANLIB=arm-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
`armv7m-uclibc`	./configure --enable-static --disable-shared \ --host=arm-linux CC=arm-linux-gcc \ LD=arm-linux-ld AR=arm-linux-ar \ RANLIB=arm-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"

arm-none-eabi-gcc	./configure --host=arm-none-eabi \ CC=arm-none-eabi-gcc LD=arm-none-eabi-ld \ AR=arm-none-eabi-ar RANLIB=arm-none-eabi-ranlib \ CFLAGS="-DNO_WOLFSSL_DIR \ -DWOLFSSL_USER_IO -DNO_WRITEV \ -mcpu=cortex-m4 -mthumb -Os \ -specs=rdimon.specs" CPPFLAGS="-I./"
mips32--glibc	./configure --host=mips-linux \ CC=mips-linux-gcc LD=mips-linux-ld \ AR=mips-linux-ar \ RANLIB=mips-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
PowerPc64le-Power8-Glibc	./configure --host=powerpc64le-buildroot-linux-gnu \ CC=powerpc64le-buildroot-linux-gnu-gcc \ LD=powerpc64le-buildroot-linux-gnu-ld \ AR=powerpc64le-buildroot-linux-gnu-ar \ RANLIB=powerpc64le-buildroot-linux-gnu-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
x86-64-core-i7-glibc	./configure --host=x86_64-linux \ CC=x86_64-linux-gcc LD=x86_64-linux-ld \ AR=x86_64-linux-ar \ RANLIB=x86_64-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"
x86-64-core-i7-musl	./configure --host=x86_64-linux \ CC=x86_64-linux-gcc LD=x86_64-linux-ld \ AR=x86_64-linux-ar \ RANLIB=x86_64-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"

x86-64-core-i7-uclibc	./configure --host=x86_64-linux \ CC=x86_64-linux-gcc LD=x86_64-linux-ld \ AR=x86_64-linux-ar \ RANLIB=x86_64-linux-ranlib \ CFLAGS="-DWOLFSSL_USER_IO -Os" \ CPPFLAGS="-I./"

Chapter 3 : Getting Started

3.1 General Description

wolfSSL, formerly CyaSSL, is about 10 times smaller than yaSSL and up to 20 times smaller than OpenSSL when using the compile options described in **Chapter 2**. User benchmarking and feedback also reports dramatically better performance from wolfSSL vs. OpenSSL in the vast majority of standard SSL operations.

For instructions on the build process please see **Chapter 2**.

3.2 Testsuite

The testsuite program is designed to test the ability of wolfSSL and its cryptography library, wolfCrypt, to run on the system.

wolfSSL needs all examples and tests to be run from the wolfSSL home directory. This is because it finds certs and keys from ./certs. To run testsuite, execute:

```
./testsuite/testsuite.test
```

or

`make test`   (when using autoconf)

On *nix or Windows the examples and testsuite will check to see if the current directory is the source directory and if so, attempt to change to the wolfSSL home directory. This should work in most setup cases, if not, just use the first method above and specify the full path.

On a successful run you should see output like this, with additional output for unit tests and cipher suite tests:

```
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES-GCM  test passed!
RANDOM   test passed!
RSA      test passed!
DH       test passed!
ECC      test passed!
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:ECDHE-
RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-
SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-
AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-
SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-
SHA384:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:DHE-RSA-
CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-OLD:ECDHE-ECDSA-CHACHA20-
POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /tmp/
output-7Iyhbo

All tests passed!
```

This indicates that everything is configured and built correctly. If any of the tests fail, make sure the build system was set up correctly. Likely culprits include having the wrong endianness or not properly setting the 64-bit type. If you've set anything to the non-default settings try removing those, rebuilding wolfSSL, and then re-testing.

3.3 Client Example

You can use the client example found in examples/client to test wolfSSL against any SSL server. To see a list of available command line runtime options, run the client with the **--help** argument:

```
./examples/client/client --help
wolfSSL client 3.15.7 NOTE: All files relative to wolfSSL home dir
Max RSA key size in bits for build is set at : 4096
-? <num>    Help, print this usage
            0: English, 1: Japanese
-h <host>   Host to connect to, default 127.0.0.1
-p <num>    Port to connect on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-V          Prints valid ssl version numbers, SSLv3(0) - TLS1.3(4)
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/client-cert.pem
-k <file>   Key file,                   default ./certs/client-key.pem
-A <file>   Certificate Authority file, default ./certs/ca-cert.pem
-Z <num>    Minimum DH key bits,        default 1024
-b <num>    Benchmark <num> connections and print stats
-L <str>    Application-Layer Protocol Negotiation ({C,F}:<list>)
-B <num>    Benchmark throughput using <num> bytes and print stats
-s          Use pre Shared keys
-d          Disable peer checks
-D          Override Date Errors example
-e          List Every cipher suite available,
-g          Send server HTTP GET
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-m          Match domain name in cert
-N          Use Non-blocking sockets
-r          Resume session
-w          Wait for bidirectional shutdown
-M <prot>   Use STARTTLS, using <prot> protocol (smtp)
-f          Fewer packets/group messages
-x          Disable client cert/key loading
-X          Driven by eXternal test case
-j          Use verify callback override
-S <str>    Use Host Name Indication
-F <num>    Use Maximum Fragment Length [1-6]
-T          Use Truncated HMAC
-n          Disable Extended Master Secret
```

```
-o          Perform OCSP lookup on peer certificate
-O <url>    Perform OCSP lookup using <url> as responder
-W <num>    Use OCSP Stapling (1 v1, 2 v2, 3 v2 multi)
-U          Atomic User Record Layer Callbacks
-P          Public Key Callbacks
-C          Disable CRL
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail]
-J          Use HelloRetryRequest to choose group for KE
-K          Key Exchange for PSK not using (EC)DHE
-I          Update keys and IVs before sending data
-y          Key Share with FFDHE named groups only
-Y          Key Share with ECC named groups only
-t          Use X25519 for key exchange
-2          Disable DH Prime check
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
```

To test against example.com:443 try the following. This is using wolfSSL compiled with the **--enable-opensslextra --enable-supportedcurves** build options:

```
./examples/client/client -h example.com -p 443 -d -g
peer's cert info:
 issuer : /C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High
Assurance Server CA
 subject: /C=US/ST=California/L=Los Angeles/O=Internet Corporation for
Assigned Names and Numbers/OU=Technology/CN=www.example.org
 altname = www.example.net
 altname = www.example.edu
 altname = www.example.com
 altname = example.org
 altname = example.net
 altname = example.edu
 altname = example.com
 altname = www.example.org
 serial number:0e:64:c5:fb:c2:36:ad:e1:4b:17:2a:eb:41:c7:8c:b0
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
Client Random :
83083A1D84404E66C86D7560A2C6ACEEEB0C35F94FDD5E07BC7507CD4E273B19
SSL connect ok, sending GET...
Server response: HTTP/1.0 200 OK
Accept-Ranges: bytes
Content-Type: text/html
Date: Tue, 20 D
ec 2016 22:52:00 GMT
Last-Modified: Tue, 20 Dec 2016 22:33:12 GMT
Server: ECS
 (pae/378A)
```

```
Content-Length: 94
Connection: close

<html><head><title>edgecastcdn.net</title></head>
<body><h1>edgecastcdn.net</h1></body></html>
```

This tells the client to connect to (-h) example.com on the HTTPS port (-p) of 443 and sends a generic (-g) GET request. The (-d) option tells the client not to verify the server. The rest is the initial output from the server that fits into the read buffer.

If no command line arguments are given, then the client attempts to connect to the localhost on the wolfSSL default port of 11111. It also loads the client certificate in case the server wants to perform client authentication.

The client is able to benchmark a connection when using the “-b <num>” argument. When used, the client attempts to connect to the specified server/port the argument number of times and gives the average time in milliseconds that it took to perform SSL_connect(). For example,

```
./examples/client/client -b 100
SSL_connect avg took: 0.653 milliseconds
```

If you'd like to change the default host from localhost, or the default port from 11111, you can change these settings in **/wolfssl/test.h**. The variables **wolfSSLIP** and **wolfSSLPort** control these settings. Re-build all of the examples including testsuite when changing these settings otherwise the test programs won't be able to connect to each other.

By default, the wolfSSL example client tries to connect to the specified server using TLS 1.2. The user is able to change the SSL/TLS version which the client uses by using the “-v” command line option. The following values are available for this option:

```
-v 0  =  SSL 3.0 (disabled by default)
-v 1  =  TLS 1.0
-v 2  =  TLS 1.1
-v 3  =  TLS 1.2 (selected by default)
-v 4  =  TLS 1.3
```

A common error users see when using the example client is -155:

```
 err = -155, ASN sig error, confirm failure
```

This is typically caused by the wolfSSL client not being able to verify the certificate of the server it is connecting to. By default, the wolfSSL client loads the yaSSL test CA certificate as a trusted root certificate. This test CA certificate will not be able to verify an external server certificate which was signed by a different CA. As such, to solve this problem, users either need to turn off verification of the peer (server), using the “-d” option:

```
./examples/client/client -h myhost.com -p 443 -d
```

Or load the correct CA certificate into the wolfSSL client using the “-A” command line option:

```
./examples/client/client -h myhost.com -p 443 -A serverCA.pem
```

Additionally, the wolfSSL example client can support the use of static memory. The wolfSSL library will need to be configured and built with the "`--enable-static`" option.

3.4 Server Example

The server example demonstrates a simple SSL server that optionally performs client authentication. Only one client connection is accepted and then the server quits. The client example in normal mode (no command line arguments) will work just fine against the example server, but if you specify command line arguments for the client example, then a client certificate isn't loaded and the wolfSSL_connect() will fail (unless client cert check is disabled using the “-d” option). The server will report an error "**-245, peer didn't send cert**". Like the example client, the server can be used with several command line arguments as well:

```
./examples/server/server --help
server 3.15.7 NOTE: All files relative to wolfSSL home dir
-? <num>    Help, print this usage
            0: English, 1: Japanese
-p <num>    Port to listen on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/server-cert.pem
-k <file>   Key file,                   default ./certs/server-key.pem
-A <file>   Certificate Authority file, default ./certs/client-cert.pem
-R <file>   Create Ready file for external monitor default none
```

```
-D <file>   Diffie-Hellman Params file, default ./certs/dh2048.pem
-Z <num>    Minimum DH key bits,        default 1024
-L <str>    Application-Layer Protocol Negotiation ({C,F}:<list>)
-d          Disable client cert check
-b          Bind to any interface instead of localhost only
-s          Use pre Shared keys
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-f          Fewer packets/group messages
-r          Allow one client Resumption
-N          Use Non-blocking sockets
-S <str>    Use Host Name Indication
-w          Wait for bidirectional shutdown
-o          Perform OCSP lookup on peer certificate
-O <url>    Perform OCSP lookup using <url> as responder
-P          Public Key Callbacks
-I          Do not send PSK identity hint
-x          Print server errors but do not close connection
-i          Loop indefinitely (allow repeated connections)
-e          Echo data mode (return raw bytes received)
-B <num>    Benchmark throughput using <num> bytes and print stats
-V          Disable CRL
-g          Return basic HTML web page
-C <num>    The number of connections to accept, default: 1
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail]
-U          Update keys and IVs before sending
-K          Key Exchange for PSK not using (EC)DHE
-y          Pre-generate Key Share using FFDHE_2048 only
-Y          Pre-generate Key Share using P-256 only
-t          Pre-generate Key share using Curve25519 only
-T          Do not generate session ticket
-2          Disable DH Prime check
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
```

3.5 EchoServer Example

The echoserver example sits in an endless loop waiting for an unlimited number of client connections. Whatever the client sends the echoserver echoes back. Client authentication isn't performed so the example client can be used against the echoserver in all 3 modes. Four special commands aren't echoed back and instruct the echoserver to take a different action.

1. "**quit**" If the echoserver receives the string "quit" it will shutdown.

2. "**break**" If the echoserver receives the string "break" it will stop the current

session but continue handling requests. This is particularly useful for DTLS testing.

3. "**printstats**" If the echoserver receives the string "printstats" it will print out statistics for the session cache.

4. "**GET**" If the echoserver receives the string "GET" it will handle it as an http get and send back a simple page with the message "greeting from wolfSSL". This allows testing of various TLS/SSL clients like Safari, IE, Firefox, gnutls, and the like against the echoserver example.

The output of the echoserver is echoed to **stdout** unless **NO_MAIN_DRIVER** is defined. You can redirect output through the shell or through the first command line argument. To create a file named output.txt with the output from the echoserver run:

```
./examples/echoserver/echoserver output.txt
```

3.6 EchoClient Example

The echoclient example can be run in interactive mode or batch mode with files. To run in interactive mode and write 3 strings "hello", "wolfssl", and "quit" results in:

```
./examples/echoclient/echoclient
hello
hello
wolfssl
wolfssl
quit
sending server shutdown command: quit!
```

To use an input file, specify the filename on the command line as the first argument. To echo the contents of the file input.txt issue:

```
./examples/echoclient/echoclient input.txt
```

If you want the result to be written out to a file, you can specify the output file name as an additional command line argument. The following command will echo the contents of file input.txt and write the result from the server to output.txt:

```
./examples/echoclient/echoclient input.txt output.txt
```

The testsuite program does just that, but hashes the input and output files to make sure that the client and server were getting/sending the correct and expected results.

3.7 Benchmark

Many users are curious about how the wolfSSL embedded SSL library will perform on a specific hardware device or in a specific environment. Because of the wide variety of different platforms and compilers used today in embedded, enterprise, and cloud-based environments, it is hard to give generic performance calculations across the board.

To help wolfSSL users and customers in determining SSL performance for wolfSSL / wolfCrypt, a benchmark application is provided which is bundled with wolfSSL. wolfSSL uses the wolfCrypt cryptography library for all crypto operations by default. Because the underlying crypto is a very performance-critical aspect of SSL/TLS, our benchmark application runs performance tests on wolfCrypt's algorithms.

The benchmark utility located in wolfcrypt/benchmark (./wolfcrypt/benchmark/benchmark) may be used to benchmark the cryptographic functionality of wolfCrypt. Typical output may look like the following (in this output, several optional algorithms/ciphers were enabled including HC-128, RABBIT, ECC, SHA-256, SHA-512, AES-GCM, AES-CCM, and Camellia):

./wolfcrypt/benchmark/benchmark

```
RNG      50 megs took 0.516 seconds,   96.975 MB/s Cycles per byte =  22.57
AES enc  50 megs took 0.278 seconds,  179.737 MB/s Cycles per byte =  12.18
AES dec  50 megs took 0.260 seconds,  192.029 MB/s Cycles per byte =  11.40
AES-GCM  50 megs took 0.840 seconds,   59.552 MB/s Cycles per byte =  36.75
AES-CCM  50 megs took 0.534 seconds,   93.548 MB/s Cycles per byte =  23.39
Camellia 50 megs took 0.376 seconds,  132.928 MB/s Cycles per byte =  16.46
HC128    50 megs took 0.032 seconds, 1550.586 MB/s Cycles per byte =   1.41
RABBIT   50 megs took 0.109 seconds,  459.559 MB/s Cycles per byte =   4.76
CHACHA   50 megs took 0.144 seconds,  347.427 MB/s Cycles per byte =   6.30
CHA-POLY 50 megs took 0.190 seconds,  262.978 MB/s Cycles per byte =   8.32
IDEA     50 megs took 0.807 seconds,   61.982 MB/s Cycles per byte =  35.31

MD5      50 megs took 0.111 seconds,  452.121 MB/s Cycles per byte =   4.84
POLY1305 50 megs took 0.039 seconds, 1281.392 MB/s Cycles per byte =   1.71
SHA      50 megs took 0.118 seconds,  424.747 MB/s Cycles per byte =   5.15
SHA-224  50 megs took 0.242 seconds,  206.789 MB/s Cycles per byte =  10.58
SHA-256  50 megs took 0.243 seconds,  206.022 MB/s Cycles per byte =  10.62
SHA-384  50 megs took 0.172 seconds,  290.787 MB/s Cycles per byte =   7.53
```

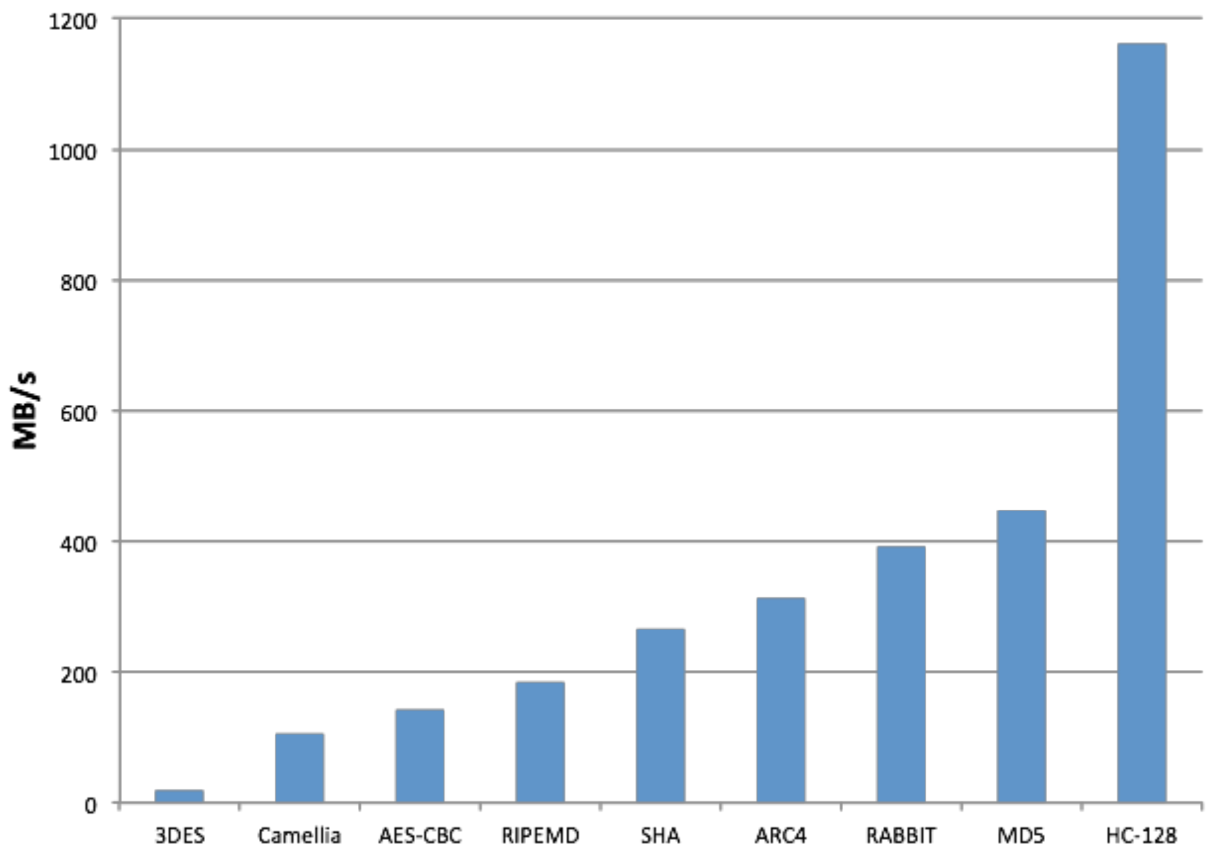
```
SHA-512  50 megs took 0.175 seconds,  286.117 MB/s Cycles per byte =   7.65

scrypt   39.698 milliseconds, avg over 10 iterations

RSA 2048 public           0.358 milliseconds, avg over 100 iterations
RSA 2048 private          4.537 milliseconds, avg over 100 iterations
DH  2048 key generation   1.391 milliseconds, avg over 100 iterations
DH  2048 key agreement    1.422 milliseconds, avg over 100 iterations

ECC  256 key generation   0.885 milliseconds, avg over 100 iterations
EC-DHE   key agreement    0.874 milliseconds, avg over 100 iterations
EC-DSA   sign   time      0.929 milliseconds, avg over 100 iterations
EC-DSA   verify time      0.602 milliseconds, avg over 100 iterations
```

This is especially useful for comparing the public key speed before and after changing the math library. You can test the results using the normal math library (**./configure**), the fastmath library (**./configure --enable-fastmath**), and the fasthugemath library (**./configure --enable-fasthugemath**).

For more details and benchmark results, please refer to the wolfSSL Benchmarks page: https://wolfssl.com/wolfSSL/benchmarks-wolfssl.html

3.7.1 Relative Performance

Although the performance of individual ciphers and algorithms will depend on the host platform, the following graph shows relative performance between wolfCrypt's ciphers. These tests were conducted on a Macbook Pro (OS X 10.6.8) running a 2.2 GHz Intel Core i7.



If you want to use only a subset of ciphers, you can customize which specific cipher suites and/or ciphers wolfSSL uses when making an SSL/TLS connection. For example, to force 128-bit AES, add the following line after the call to wolfSSL_CTX_new (SSL_CTX_new):

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

3.7.2 Benchmarking Notes

1. The benchmark application can be used to test individual algorithms by passing the algorithm as an option to the benchmark utility (**-<alrogithmname>**). The current list of supported algorithms that the benchmark can test in version 4.0.0 includes the following:
   `cipher aes-cbc aes-gcm aes-ecb aes-xts aes-cfb aes-ctr aes-ccm camellia arc4 hc128 rabbit chacha20 chacha20-poly1305`

```
des idea digest md5 poly1305 sha sha2 sha224 sha256 sha384
sha512 sha3 sha3-224 sha3-256 sha3-384 sha3-512 ripemd mac
cmac hmac hmac-md5 hmac-sha hmac-sha224 hmac-sha256 hmac-
sha384 hmac-sha512 asym rsa-kg rsa rsa-sz dh ecc-kg ecc
ecc-enc curve25519_kg x25519 ed25519-kg ed25519 other rng
scrypt
```

2. The benchmark block size can be selected by passing the desired block size (in bytes) to the benchmark script (**./examples/benchmark/benchmark <block-size>**).

3. Benchmarking output can also be viewed in logarithmic form as well (**-base10**)

4. The processors **native register size** (32 vs 64-bit) can make a big difference when doing 1000+ bit public key operations.

5. **keygen** (--enable-keygen) will allow you to also benchmark key generation speeds when running the benchmark utility.

6. **fastmath** (--enable-fastmath) reduces dynamic memory usage and speeds up public key operations. If you are having trouble building on 32-bit platform with fastmath, disable shared libraries so that PIC isn't hogging a register (also see notes in the README)

   ```
   ./configure --enable-fastmath --disable-shared
   make clean
   make
   ```

   ***Note:** doing a "make clean" is good practice with wolfSSL when switching configure options.

7. By default, fastmath tries to use assembly optimizations if possible. If assembly optimizations don't work, you can still use fastmath without them by adding

TFM_NO_ASM to CFLAGS when building wolfSSL:

```
./configure --enable-fastmath C_EXTRA_FLAGS="-DTFM_NO_ASM"
```

8. Using fasthugemath can try to push fastmath even more for users who are not running on embedded platforms:

```
./configure --enable-fasthugemath
```

9. With the default wolfSSL build, we have tried to find a good balance between memory usage and performance. If you are more concerned about one of the two, please refer back to **Chapter 2** for additional wolfSSL configuration options.

10. **Bulk Transfers:** wolfSSL by default uses 128 byte I/O buffers since about 80% of SSL traffic falls within this size and to limit dynamic memory use. It can be configured to use 16K buffers (the maximum SSL size) if bulk transfers are required.

3.7.3 Benchmarking on Embedded Systems

There are several build options available to make building the benchmark application on an embedded system easier. These include:

BENCH_EMBEDDED - enabling this define will switch the benchmark application from using Megabytes to using Kilobytes, therefore reducing the memory usage. By default, when using this define, ciphers and algorithms will be benchmarked with 25kB. Public key algorithms will only be benchmarked over 1 iteration (as public key operations on some embedded processors can be fairly slow). These can be adjusted in **benchmark.c** by altering the variables "numBlocks" and "times" located inside the BENCH_EMBEDDED define.

USE_CERT_BUFFERS_1024 - enabling this define will switch the benchmark application from loading test keys and certificates from the file system and instead use 1024-bit key and certificate buffers located in <wolfssl_root>/wolfssl/certs_test.h. It is useful to use this define when an embedded platform has no filesystem (used with

NO_FILESYSTEM) and a slow processor where 2048-bit public key operations may not be reasonable.

USE_CERT_BUFFERS_2048 - enabling this define is similar to USE_CERT_BUFFERS_1024 accept that 2048-bit key and certificate buffers are used instead of 1024-bit ones. This define is useful when the processor is fast enough to do 2048-bit public key operations but when there is no filesystem available to load keys and certificates from files.

3.8 Changing a Client Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a client application, using the wolfSSL native API. For a server explanation, please see **Section 3.9**. A more complete walk-through with example code is located in the SSL Tutorial in **Chapter 11**. If you want more information about the OpenSSL compatibility layer, please see **Chapter 13**.

1. Include the wolfSSL header

   ```
   #include <wolfssl/ssl.h>
   ```

2. Change all calls from read() (or recv()) to wolfSSL_read() so

   ```
   result = read(fd, buffer, bytes);
   ```

   becomes

   ```
   result = wolfSSL_read(ssl, buffer, bytes);
   ```

3. Change all calls from write (or send) to wolfSSL_write() so

   ```
   result = write(fd, buffer, bytes);
   ```

   becomes

   ```
   result = wolfSSL_write(ssl, buffer, bytes);
   ```

4. You can manually call wolfSSL_connect() but that's not even necessary; the first call to wolfSSL_read() or wolfSSL_write() will initiate the wolfSSL_connect() if it hasn't taken place yet.

5. Initialize wolfSSL and the WOLFSSL_CTX. You can use one WOLFSSL_CTX no matter how many WOLFSSL objects you end up creating. Basically you'll just need to load CA certificates to verify the server you are connecting to. Basic initialization looks like:

```
wolfSSL_Init();

WOLFSSL_CTX* ctx;

if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_client_method())) == NULL)
{
    fprintf(stderr, "wolfSSL_CTX_new error.\n");
    exit(EXIT_FAILURE);
}

if (wolfSSL_CTX_load_verify_locations(ctx,"./ca-cert.pem",0) !=
    SSL_SUCCESS) {
    fprintf(stderr, "Error loading ./ca-cert.pem,"
                    " please check the file.\n");
    exit(EXIT_FAILURE);
}
```

6. Create the WOLFSSL object after each TCP connect and associate the file descriptor with the session:

```
/*after connecting to socket fd*/

WOLFSSL* ssl;

if ( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, fd);
```

7. Error checking. Each wolfSSL_read() and wolfSSL_write() call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like read() and write(). In the event of an error you can use two calls to get more information about the error:

```
char errorString[80];
int err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non-blocking sockets, you can test for errno EAGAIN/ EWOULDBLOCK or more correctly you can test the specific error code returned by wolfSSL_get_error() for **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**.

8. Cleanup. After each WOLFSSL object is done being used you can free it up by calling:

```
wolfSSL_free(ssl);
```

When you are completely done using SSL/TLS altogether you can free the WOLFSSL_CTX object by calling:

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

For an example of a client application using wolfSSL, see the client example located in the <wolfssl_root>/examples/client.c file.

3.9 Changing a Server Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a server application using the wolfSSL native API. For a client explanation, please see **section 3.8**. A more complete walk-through, with example code, is located in the SSL Tutorial in **Chapter 11**.

1. Follow the instructions above for a client, except change the client method call in step 5 to a server one, so

```
wolfSSL_CTX_new(wolfTLSv1ls_client_method())
```

becomes

```
wolfSSL_CTX_new(wolfTLSv1_server_method())
```

or even

```
wolfSSL_CTX_new(wolfSSLv23_server_method())
```

To allow SSLv3 and TLSv1+ clients to connect to the server.

2. Add the server's certificate and key file to the initialization in step 5 above:

```
if (wolfSSL_CTX_use_certificate_file(ctx,"./server-cert.pem",
    SSL_FILETYPE_PEM) != SSL_SUCCESS) {
    fprintf(stderr, "Error loading ./server-cert.pem,"
                    " please check the file.\n");
    exit(EXIT_FAILURE);
}

if (wolfSSL_CTX_use_PrivateKey_file(ctx,"./server-key.pem",
    SSL_FILETYPE_PEM) != SSL_SUCCESS) {
    fprintf(stderr, "Error loading ./server-key.pem,"
                    " please check the file.\n");
    exit(EXIT_FAILURE);
}
```

It is possible to load certificates and keys from buffers as well if there is no filesystem available. In this case, see the wolfSSL_CTX_use_certificate_buffer() and wolfSSL_CTX_use_PrivateKey_buffer() API documentation, linked here, for more information.

For an example of a server application using wolfSSL, see the server example located in the <wolfssl_root>/examples/server.c file.

Chapter 4: Features

wolfSSL (formerly CyaSSL) supports the C programming language as a primary interface, but also supports several other host languages, including Java, PHP, Perl, and Python (through a SWIG interface). If you have interest in hosting wolfSSL in another programming language that is not currently supported, please contact us.

This chapter covers some of the features of wolfSSL in more depth, including Stream Ciphers, AES-NI, IPv6 support, SSL Inspection (Sniffer) support, and more.

4.1 Features Overview

For an overview of wolfSSL features, please reference the wolfSSL product webpage: https://wolfssl.com/wolfSSL/Products-wolfssl.html

4.2 Protocol Support

wolfSSL supports **SSL 3.0**, **TLS** (**1.0**, **1.1**, **1.2, 1.3**), and **DTLS** (**1.0** and **1.2**). You can easily select a protocol to use by using one of the following functions (as shown for either the client or server). wolfSSL does not support SSL 2.0, as it has been insecure for several years. The client and server functions below change slightly when using the OpenSSL compatibility layer. For the OpenSSL-compatible functions, please see **Chapter 13**.

4.2.1 Server Functions

```
wolfDTLSv1_server_method(void);       /*DTLS 1.0*/
wolfDTLSv1_2_server_method(void);   /*DTLS 1.2*/
wolfSSLv3_server_method(void);        /*SSL 3.0*/
wolfTLSv1_server_method(void);        /*TLS 1.0*/
wolfTLSv1_1_server_method(void); /*TLS 1.1*/
wolfTLSv1_2_server_method(void); /*TLS 1.2*/
wolfTLSv1_3_server_method(void); /*TLS 1.3*/
wolfSSLv23_server_method(void);       /*Use highest possible version
                                         from SSLv3 - TLS 1.2*/
```

wolfSSL supports robust server downgrade with the **wolfSSLv23_server_method()** function. See section 4.2.3 for a details.

4.2.2 Client Functions

```
wolfDTLSv1_client_method(void);       /* DTLS 1.0*/
wolfDTLSv1_2_client_method(void);   /* DTLS 1.2*/
wolfSSLv3_client_method(void);        /* SSL 3.0*/
wolfTLSv1_client_method(void);        /* TLS 1.0*/
wolfTLSv1_1_client_method(void); /* TLS 1.1*/
wolfTLSv1_2_client_method(void); /* TLS 1.2*/
```

```
wolfTLSv1_3_client_method(void); /* TLS 1.3*/
wolfSSLv23_client_method(void);        /* Use highest possible version
                                           from SSLv3 - TLS 1.2*/
```

wolfSSL supports robust client downgrade with the **wolfSSLv23_client_method()** function. See section 4.2.3 for a details.

For details on how to use these functions, please see **Chapter 3**, “Getting Started.” For a comparison between SSL 3.0, TLS 1.0, 1.1, 1.2, and DTLS, please see Appendix A.

4.2.3 Robust Client and Server Downgrade

Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLS 1.0 and tries to connect to an SSL 3.0 only server, the connection will fail, likewise connecting to a TLS 1.1 will fail as well.

To resolve this issue, a client that uses the **wolfSSLv23_client_method()** function will use the highest protocol version supported by the server and downgrade to TLS 1.0 if needed. In this case, the client will be able to connect to a server running TLS 1.0 - TLS 1.2. The only versions it can't connect to is SSL 2.0 which has been insecure for years, and SSL 3.0 which has been disabled by default.

Similarly, a server using the **wolfSSLv23_server_method()** function can handle clients supporting protocol versions from TLS 1.0 - TLS 1.2. A wolfSSL server can't accept a connection from SSLv2 because no security is provided.

4.2.4 IPv6 Support

If you are an adopter of IPv6 and want to use an embedded SSL implementation then you may have been wondering if wolfSSL supports IPv6. The answer is yes, we do support wolfSSL running on top of IPv6.

wolfSSL was designed as IP neutral, and will work with both IPv4 and IPv6, but the current test applications default to IPv4 (so as to apply to a broader range of systems). To change the test applications to IPv6, use the **--enable-ipv6** option while building wolfSSL.

Further information on IPv6 can be found here:
http://en.wikipedia.org/wiki/IPv6.

4.2.5 DTLS

wolfSSL has support for **DTLS** ("Datagram" TLS) for both client and server. The current supported version is DTLS 1.0.

The TLS protocol was designed to provide a secure transport channel across a **reliable** medium (such as TCP). As application layer protocols began to be developed using UDP transport (such as SIP and various electronic gaming protocols), a need arose for a way to provide communications security for applications which are delay sensitive. This need lead to the creation of the DTLS protocol.

Many people believe the difference between TLS and DTLS is the same as TCP vs. UDP. This is incorrect. UDP has the benefit of having no handshake, no tear-down, and no delay in the middle if something gets lost (compared with TCP). DTLS on the other hand, has an extended SSL handshake and tear-down and must implement TCP-like behavior for the handshake. In essence, DTLS reverses the benefits that are offered by UDP in exchange for a secure connection.

DTLS can be enabled when building wolfSSL by using the **--enable-dtls** build option.

4.2.6 LwIP (Lightweight Internet Protocol)

wolfSSL supports the lightweight internet protocol implementation out of the box. To use this protocol all you need to do is define WOLFSSL_LWIP or navigate to the **settings.h** file and uncomment the line:

```
/*#define WOLFSSL_LWIP*/
```

The focus of lwIP is to reduce RAM usage while still providing a full TCP stack. That focus makes lwIP great for use in embedded systems, an area where wolfSSL is an ideal match for SSL/TLS needs.

4.2.7 TLS Extensions

A list of TLS extensions supported by wolfSSL and note of which RFC can be referenced for the given extension.

RFC	Extension	wolfSSL Type
6066	Server Name Indication	TLSX_SERVER_NAME
6066	Maximum Fragment Length Negotiation	TLSX_MAX_FRAGMENT_LENGTH
6066	Truncated HMAC	TLSX_TRUNCATED_HMAC
6066	Status Request	TLSX_STATUS_REQUEST
7919	Supported Groups	TLSX_SUPPORTED_GROUPS
4492	EC Point Formats	TLSX_EC_POINT_FORMATS
5246	Signature Algorithm	TLSX_SIGNATURE_ALGORITHMS
7301	Application Layer Protocol Negotiation	TLSX_APPLICATION_LAYER_PROTOCOL
6961	Multiple Certificate Status Request	TLSX_STATUS_REQUEST_V2
Draft	Quantum-Safe Hybrid Key Exchange	TLSX_QUANTUM_SAFE_HYBRID
5077	Session Ticket	TLSX_SESSION_TICKET
5746	Renegotiation Indication	TLSX_RENEGOTIATION_INFO
Draft	Key Share	TLSX_KEY_SHARE
Draft	Pre Shared Key	TLSX_PRE_SHARED_KEY
Draft	PSK Key Exchange Modes	TLSX_PSK_KEY_EXCHANGE_MODES
Draft	Early Data	TLSX_EARLY_DATA
Draft	Cookie	TLSX_COOKIE
Draft	Supported Versions	TLSX_SUPPORTED_VERSIONS
Draft	Post Handshake Authorization	TLSX_POST_HANDSHAKE_AUTH

4.3 Cipher Support

4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes

To see what ciphers are currently being used you can call the method:

```
wolfSSL_get_ciphers()
```

This function will return the currently enabled cipher suites.

Cipher suites come in a variety of strengths. Because they are made up of several different types of algorithms (authentication, encryption, and message authentication code (MAC)), the strength of each varies with the chosen key sizes.
There can be many methods of grading the strength of a cipher suite - the specific method used seems to vary between different projects and companies and can include things such as symmetric and public key algorithm key sizes, type of algorithm, performance, and known weaknesses.

NIST (National Institute of Standards and Technology) makes recommendations on choosing an acceptable cipher suite by providing comparable algorithm strengths for varying key sizes of each. The strength of a cryptographic algorithm depends on the algorithm and the key size used. The NIST Special Publication, SP800-57, states that two algorithms are considered to be of comparable strength as follows:

> … two algorithms are considered to be of comparable strength for the given key sizes (X and Y) if the amount of work needed to "break the algorithms" or determine the keys (with the given key sizes) is approximately the same using a given resource. The security strength of an algorithm for a given key size is traditionally described in terms of the amount of work it takes to try all keys for a symmetric algorithm with a key size of "X" that has no shortcut attacks (i.e., the most efficient attack is to try all possible keys).

The following two tables are based off of both Table 2 (pg. 64) and Table 4 (pg. 66) from NIST SP800-57, and shows comparable security strength between algorithms as well as a strength measurement (based off of NIST's suggested algorithm security lifetimes using bits of security).

Note: In the following table "L" is the size of the public key for finite field cryptography (FFC), "N" is the size of the private key for FFC, "k" is considered the key size for integer factorization cryptography (IFC), and "f" is considered the key size for elliptic curve cryptography.

Bits of Security	**Symmetric Key Algorithms**	**FFC** Key Size (DSA, DH, etc.)	**IFC** Key Size (RSA, etc.)	**ECC** Key Size (ECDSA, etc.)
80	2TDEA, etc.	L = 1024 N = 160	k = 1024	f = 160-223
128	AES-128, etc.	L = 3072 N = 256	k = 3072	f = 256-383
192	AES-192, etc.	L = 7680 N = 384	k = 7680	f = 384-511
256	AES-256, etc.	L = 15360 N = 512	k = 15360	f = 512+

(Table 2: Relative Bit and Key Strengths)

Bits of Security	**Description**
80	Security good through 2010
128	Security good through 2030
192	Long Term Protection
256	Secure for the foreseeable future

(Table 3: Bit Strength Descriptions)

Using this table as a guide, to begin to classify a cipher suite, we categorize it based on the strength of the symmetric encryption algorithm. In doing this, a rough grade classification can be devised to classify each cipher suite based on bits of security (only taking into account symmetric key size):

LOW = bits of security smaller than 128 bits
MEDIUM = bits of security equal to 128 bits
HIGH = bits of security larger than 128 bits

Outside of the symmetric encryption algorithm strength, the strength of a cipher suite will depend greatly on the key sizes of the key exchange and authentication algorithm keys. The strength is only as good as the cipher suite's weakest link.

Following the above grading methodology (and only basing it on symmetric encryption algorithm strength), wolfSSL 2.0.0 currently supports a total of 0 LOW strength cipher suites, 12 MEDIUM strength cipher suites, and 8 HIGH strength cipher suites – as listed below. The following strength classification could change depending on the chosen key sizes of the other algorithms involved. For a reference on hash function security strength, see Table 3 (pg. 64) of NIST SP800-57.

In some cases, you will see ciphers referenced as "**EXPORT**" ciphers. These ciphers originated from the time period in US history (as late as 1992) when it was illegal to export software with strong encryption from the United States. Strong encryption was classified as "Munitions" by the US Government (under the same category as Nuclear Weapons, Tanks, and Ballistic Missiles). Because of this restriction, software being exported included "weakened" ciphers (mostly in smaller key sizes). In the current day, this restriction has been lifted, and as such, EXPORT ciphers are no longer a mandated necessity.

4.3.2 Supported Cipher Suites

The following cipher suites are supported by wolfSSL. A cipher suite is a combination of authentication, encryption, and message authentication code (MAC) algorithms which are used during the TLS or SSL handshake to negotiate security settings for a connection.

Each cipher suite defines a key exchange algorithm, a bulk encryption algorithm, and a message authentication code algorithm (MAC). The **key exchange algorithm** (RSA, DSS, DH, EDH) determines how the client and server will authenticate during the handshake process. The **bulk encryption algorithm** (DES, 3DES, AES, ARC4, RABBIT, HC-128), including block ciphers and stream ciphers, is used to encrypt the message stream. The **message authentication code (MAC) algorithm** (MD2, MD5, SHA-1, SHA-256, SHA-512, RIPEMD) is a hash function used to create the message digest.

The table below matches up to the cipher suites (and categories) found in <wolfssl_root>/wolfssl/internal.h (starting at about line 706). If you are looking for a cipher suite which is not in the following list, please contact us to discuss getting it added to wolfSSL.

wolfSSL Cipher Suites (version 3.10.0)	

TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA TLS_DHE_RSA_WITH_AES_256_CBC_SHA TLS_DHE_RSA_WITH_AES_128_CBC_SHA TLS_DH_anon_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_NULL_SHA TLS_PSK_WITH_AES_256_CBC_SHA TLS_PSK_WITH_AES_128_CBC_SHA256 TLS_PSK_WITH_AES_256_CBC_SHA384 TLS_PSK_WITH_AES_128_CBC_SHA TLS_PSK_WITH_NULL_SHA256 TLS_PSK_WITH_NULL_SHA384 TLS_PSK_WITH_NULL_SHA SSL_RSA_WITH_RC4_128_SHA SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_3DES_EDE_CBC_SHA SSL_RSA_WITH_IDEA_CBC_SHA	
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_RC4_128_SHA TLS_ECDHE_ECDSA_WITH_RC4_128_SHA TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 TLS_ECDHE_PSK_WITH_NULL_SHA256 TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256 TLS_ECDHE_ECDSA_WITH_NULL_SHA	ECC cipher suites

TLS_ECDH_RSA_WITH_AES_256_CBC_SHA TLS_ECDH_RSA_WITH_AES_128_CBC_SHA TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA TLS_ECDH_RSA_WITH_RC4_128_SHA TLS_ECDH_ECDSA_WITH_RC4_128_SHA TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256 TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384 TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	Static ECDH cipher suites
TLS_RSA_WITH_HC_128_MD5 TLS_RSA_WITH_HC_128_SHA TLS_RSA_WITH_RABBIT_SHA	wolfSSL extension - eSTREAM cipher suites
TLS_RSA_WITH_AES_128_CBC_B2B256 TLS_RSA_WITH_AES_256_CBC_B2B256 TLS_RSA_WITH_HC_128_B2B256	Blake2b cipher suites
TLS_QSH	wolfSSL extension - Quantum-Safe Handshake
TLS_NTRU_RSA_WITH_RC4_128_SHA TLS_NTRU_RSA_WITH_3DES_EDE_CBC_SHA TLS_NTRU_RSA_WITH_AES_128_CBC_SHA TLS_NTRU_RSA_WITH_AES_256_CBC_SHA	wolfSSL extension - NTRU cipher suites
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA256 TLS_RSA_WITH_AES_128_CBC_SHA256 TLS_RSA_WITH_NULL_SHA256 TLS_DHE_PSK_WITH_AES_128_CBC_SHA256 TLS_DHE_PSK_WITH_NULL_SHA256	SHA-256 cipher suites
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384 TLS_DHE_PSK_WITH_NULL_SHA384	SHA-384 cipher suites

TLS_RSA_WITH_AES_128_GCM_SHA256 TLS_RSA_WITH_AES_256_GCM_SHA384 TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 TLS_PSK_WITH_AES_128_GCM_SHA256 TLS_PSK_WITH_AES_256_GCM_SHA384 TLS_DHE_PSK_WITH_AES_128_GCM_SHA256 TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	AES-GCM cipher suites
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256 TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	ECC AES-GCM cipher suites
TLS_RSA_WITH_AES_128_CCM_8 TLS_RSA_WITH_AES_256_CCM_8 TLS_ECDHE_ECDSA_WITH_AES_128_CCM TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8 TLS_PSK_WITH_AES_128_CCM TLS_PSK_WITH_AES_256_CCM TLS_PSK_WITH_AES_128_CCM_8 TLS_PSK_WITH_AES_256_CCM_8 TLS_DHE_PSK_WITH_AES_128_CCM TLS_DHE_PSK_WITH_AES_256_CCM	AES-CCM cipher suites
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA TLS_RSA_WITH_CAMELLIA_256_CBC_SHA TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256 TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256 TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256 TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	Camellia cipher suites

TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256 TLS_PSK_WITH_CHACHA20_POLY1305_SHA256 TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256 TLS_ECDHE_ECDSA_WITH_CHACHA20_OLD_POLY1305_SHA256 TLS_DHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256	ChaCha cipher suites
TLS_EMPTY_RENEGOTIATION_INFO_SCSV	Renegotiation Indication Extension Special Suite

(Table 4: wolfSSL Cipher Suites)

4.3.3 AEAD Suites

wolfSSL supports AEAD suites, including AES-GCM, AES-CCM, and CHACHA-POLY1305. The big difference between these AEAD suites and others is that they authenticate the encrypted data. This helps with mitigating man in the middle attacks that result in having data tampered with. AEAD suites use a combination of a block cipher (or more recently also a stream cipher) algorithm combined with a tag produced by a keyed hash algorithm. Combining these two algorithms is handled by the wolfSSL encrypt and decrypt process which makes it easier for users. All that is needed for using a specific AEAD suite is simply enabling the algorithms that are used in a supported suite.

4.3.4 Block and Stream Ciphers

wolfSSL supports the **AES**, **DES**, **3DES**, and **Camellia** block ciphers and the **RC4**, **RABBIT**, **HC-128** and **CHACHA20** stream ciphers. AES, DES, 3DES, RC4 and RABBIT are enabled by default. Camellia, HC-128, and ChaCha20 can be enabled when building wolfSSL (with the **--enable-hc128**, **--enable-camellia**, and **--enable-chacha** build options, respectively). The default mode of AES is CBC mode. To enable GCM or

CCM mode with AES, use the **--enable-aesgcm** and **--enable-aesccm** build options. Please see the examples for usage and the wolfCrypt Usage Reference (**Chapter 10**) for specific usage information.

SSL uses RC4 as the default stream cipher. It's a good one, though it's getting a little old. wolfSSL has added two ciphers from the eStream project into the code base, RABBIT and HC-128. RABBIT is nearly twice as fast as RC4 and HC-128 is about 5 times as fast! So if you've ever decided not to use SSL because of speed concerns, using wolfSSL's stream ciphers should lessen or eliminate that performance doubt. Recently wolfSSL also added ChaCha20. While RC4 is about 11% more performant than ChaCha, RC4 is generally considered less secure than ChaCha. ChaCha can put up very nice times of it’s own with added security as a tradeoff.

To see a comparison of cipher performance, visit the wolfSSL Benchmark web page, located here: http://wolfssl.com/yaSSL/benchmarks-wolfssl.html.

4.3.4.1 What’s the Difference?

A block cipher has to be encrypted in chunks that are the block size for the cipher. For example, AES has block size of 16 bytes. So if you're encrypting a bunch of small, 2 or 3 byte chunks back and forth, over 80% of the data is useless padding, decreasing the speed of the encryption/decryption process and needlessly wasting network bandwidth to boot. Basically block ciphers are designed for large chunks of data, have block sizes requiring padding, and use a fixed, unvarying transformation.

Stream ciphers work well for large or small chunks of data. They are suitable for smaller data sizes because no block size is required. If speed is a concern, stream ciphers are your answer, because they use a simpler transformation that typically involves an xor'd keystream. So if you need to stream media, encrypt various data sizes including small ones, or have a need for a fast cipher then stream ciphers are your best bet.

4.3.5 Hashing Functions

wolfSSL supports several different hashing functions, including **MD2**, **MD4**, **MD5**, **SHA-1**, **SHA-2** (SHA-224, SHA-256, SHA-384, SHA-512**)**, **SHA-3** (BLAKE2), and **RIPEMD-160**. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Section 10.1.

4.3.6 Public Key Options

wolfSSL supports the **RSA**, **ECC**, **DSA/DSS**, **DH**, and **NTRU** public key options, with support for **EDH** (Ephemeral Diffie-Hellman) on the wolfSSL server. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, section 10.5.

wolfSSL has support for four cipher suites utilizing NTRU public key:

> **TLS_NTRU_RSA_WITH_3DES_EDE_CBC_SHA**
> **TLS_NTRU_RSA_WITH_RC4_128_SHA**
> **TLS_NTRU_RSA_WITH_AES_128_CBC_SHA**
> **TLS_NTRU_RSA_WITH_AES_256_CBC_SHA**

The strongest one, AES-256, is the default. If wolfSSL is enabled with NTRU and the NTRU library is available, these cipher suites are built into the wolfSSL library. A wolfSSL client will have these cipher suites available without any interaction needed by the user. On the other hand, a wolfSSL server application will need to load an NTRU private key and NTRU x509 certificate in order for those cipher suites to be available for use.

The example servers, echoserver and server, both use the define HAVE_NTRU (which is turned on by enabling NTRU) to specify whether or not to load NTRU keys and certificates. The wolfSSL package comes with test keys and certificates in the /certs directory. ntru-cert.pem is the certificate and ntru-key.raw is the private key blob.

The wolfSSL NTRU cipher suites are given the highest preference order when the protocol picks a suite. Their exact preference order is the reverse of the above listed suites, i.e., AES-256 will be picked first and 3DES last before moving onto the "standard" cipher suites. Basically, if a user builds NTRU into wolfSSL and both sides of the connection support NTRU then an NTRU cipher suite will be picked unless a user on one side has explicitly excluded them by stating to only use different cipher suites. Using NTRU over RSA can provide a 20 - 200X speed improvement. The improvement increases as the size of keys increases, meaning a much larger speed benefit when using large keys (8192-bit) versus smaller keys (1024-bit).

4.3.7 ECC Support

wolfSSL has support for Elliptic Curve Cryptography (ECC) including but not limited to:

ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-PSK and ECDHE-RSA.

wolfSSL's ECC implementation can be found in the **<wolfssl_root>/wolfssl/wolfcrypt/ecc.h** header file and the **<wolfssl_root>/wolfcrypt/src/ecc.c** source file.

Supported cipher suites are shown in the table above. ECC is disabled by default on non x86_64 builds, but can be turned on when building wolfSSL with the HAVE_ECC define or by using the autoconf system:

```
./configure --enable-ecc
make
make check
```

When "make check" runs, note the numerous cipher suites that wolfSSL checks (if make check doesn't produce a list of cipher suites run ./testsuite/testsuite.test on its own). Any of these cipher suites can be tested individually, e.g., to try ECDH-ECDSA with AES256-SHA, the example wolfSSL server can be started like this:

```
./examples/server/server -d -l ECDHE-ECDSA-AES256-SHA -c ./
certs/server-ecc.pem -k ./certs/ecc-key.pem
```

(-d) disables client cert check while (-l) specifies the cipher suite list. (-c) is the certificate to use and (-k) is the corresponding private key to use. To have the client connect try:

```
./examples/client/client -A ./certs/server-ecc.pem
```

where (-A) is the CA certificate to use to verify the server.

4.3.8 PKCS Support

PKCS (Public Key Cryptography Standards) refers to a group of standards created and published by RSA Security, Inc. wolfSSL has support for **PKCS #1, PKCS #3, PKCS #5, PKCS #7, PKCS #8, PKCS #9, PKCS #10, PKCS #11,** and **PKCS #12.**

Additionally, wolfSSL also provides support for RSA-Probabilistic Signature Scheme (PSS), which is standardized as part of PKCS #1.

4.3.8.1 PKCS #5, PBKDF1, PBKDF2, PKCS #12

PKCS #5 is a password based key derivation method which combines a password, a salt, and an iteration count to generate a password-based key. wolfSSL supports both PBKDF1 and PBKDF2 key derivation functions. A key derivation function produces a derived key from a base key and other parameters (such as the salt and iteration count as explained above). PBKDF1 applies a hash function (MD5, SHA1, etc) to derive keys, where the derived key length is bounded by the length of the hash function output. With PBKDF2, a pseudorandom function is applied (such as HMAC-SHA-1) to derive the keys. In the case of PBKDF2, the derived key length is unbounded.

wolfSSL also supports the PBKDF function from PKCS #12 in addition to PBKDF1 and PBKDF2. The function prototypes look like this:

```
int PBKDF2(byte* output, const byte* passwd, int pLen,
           const byte* salt,int sLen, int iterations,
           int kLen, int hashType);

int PKCS12_PBKDF(byte* output, const byte* passwd, int pLen,
                 const byte* salt, int sLen, int iterations,
                 int kLen, int hashType, int purpose);
```

output contains the derived key, **passwd** holds the user password of length **pLen**, **salt** holds the salt input of length **sLen**, **iterations** is the number of iterations to perform, **kLen** is the desired derived key length, and **hashType** is the hash to use (which can be MD5, SHA1, or SHA2).

If you are using `./configure` to build wolfssl, the way to enable this functionality is to use the option `--enable-pwdbased`

A full example can be found in <**wolfSSL Root**>**/wolfcrypt/test.c**. More information can be found on PKCS #5, PBKDF1, and PBKDF2 from the following specifications:

PKCS#5, PBKDF1, PBKDF2: http://tools.ietf.org/html/rfc2898

4.3.8.2 PKCS #7

PKCS#7 is used to sign, encrypt, or decrypt messages under Public Key Infrastructure (PKI). It is also used for certificate dissemination, but is most commonly used for single sign-on.

wolfSSL provides support for Cryptographic Message Syntax (CMS) as part of PKCS#7 support. wolfSSL also provides support for using PKCS#7 with Key-Encryption Key Recipient Info (KEKRI), Password Recipient Info (PWRI), and Other Recipient Info (ORI) types.

wolfSSL PKCS#7 support can be enabled by using the **--enable-pkcs7** configure option.

When PKCS#7 is also used with hardware-based private keys, wolfSSL also provides support for devId use. devId can be set through calls to "`wolfSSL_SetDevId`" and "`wolfSSL_CTX_SetDevId`".

wolfSSL provides support for the combination of PKCS#7 with static memory as well. Static memory support can be enabled by defining the `WOLFSSL_STATIC_MEMORY` option or adding the option `--enable-staticmemory` to the wolfSSL configure script.

Note: wolfSSL PKCS#7 BER encodings have an indefinite max length, which are limited to the size constraints of the device.

PKCS#7: https://tools.ietf.org/html/rfc2315

4.3.8.3 PKCS #8

PKCS #8 is designed as the Private-Key Information Syntax Standard, which is used to store private key information - including a private key for some public-key algorithm and set of attributes.

The PKCS #8 standard has two versions which describe the syntax to store both encrypted private keys and non-encrypted keys. wolfSSL supports both unencrypted and encrypted PKCS #8. Supported formats include PKCS #5 version 1 - version 2, and PKCS#12. Types of encryption available include DES, 3DES, RC4, and AES.

PKCS#8: http://tools.ietf.org/html/rfc5208

4.3.9 Forcing the Use of a Specific Cipher

By default, wolfSSL will pick the "best" (highest security) cipher suite that both sides of the connection can support. To force a specific cipher, such as 128 bit AES, add something similar to:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

after the call to wolfSSL_CTX_new(); so that you have:

```
ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

4.3.10 Quantum-Safe Handshake Ciphersuite

wolfSSL has support for the cipher suite utilizing post quantum handshake cipher suite such as with NTRU:

TLS_QSH

If wolfSSL is enabled with NTRU and the NTRU package is available, the TLS_QSH cipher suite is built into the wolfSSL library. A wolfSSL client and server will have this cipher suite available without any interaction needed by the user.

The wolfSSL quantum safe handshake ciphersuite is given the highest preference order when the protocol picks a suite. Basically, if a user builds NTRU into wolfSSL and both sides of the connection support NTRU then an NTRU cipher suite will be picked unless a user on one side has explicitly excluded them by stating to only use different cipher suites.

Users can adjust what crypto algorithms and if the client sends across public keys by using the function examples

```
wolfSSL_UseClientQSHKeys(ssl, 1);

wolfSSL_UseSupportedQSH(ssl, WOLFSSL_NTRU_EESS439);
```

To test if a QSH connection was established after a client has connected the following function example can be used.

```
wolfSSL_isQSH(ssl);
```

4.4 Hardware Accelerated Crypto

wolfSSL is able to take advantage of several hardware accelerated (or "assisted") crypto functionalities in various processors and chips. The following sections explain which technologies wolfSSL supports out-of-the-box.

4.4.1 AES-NI

AES is a key encryption standard used by governments worldwide, which wolfSSL has always supported. Intel has released a new set of instructions that is a faster way to implement AES. wolfSSL is the first SSL library to fully support the new instruction set for production environments.

Essentially, Intel and AMD have added AES instructions at the chip level that perform the computationally-intensive parts of the AES algorithm, boosting performance. For a list of Intel's chips that currently have support for AES-NI, you can look here:

http://ark.intel.com/search/advanced/?s=t&AESTech=true

We have added the functionality to wolfSSL to allow it to call the instructions directly from the chip, instead of running the algorithm in software. This means that when you're running wolfSSL on a chipset that supports AES-NI, you can run your AES crypto 5-10 times faster!

If you are running on an AES-NI supported chipset, enable AES-NI with the **--enable-aesni** build option. To build wolfSSL with AES-NI, GCC 4.4.3 or later is required to make use of the assembly code. wolfSSL supports the ASM instructions on AMD processors using the same build options.

References and further reading on AES-NI, ordered from general to specific, are listed below. For information about performance gains with AES-NI, please see the third link to the Intel Software Network page.

AES (Wikipedia)	http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
AES-NI (Wikipedia)	http://en.wikipedia.org/wiki/AES_instruction_set
AES-NI (Intel Software Network page)	http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/

4.4.2 STM32F1, STM32F2, STM32F7, STM32L4

wolfSSL is able to use the STM32 hardware-based cryptography and random number generator through the STM32 Standard Peripheral Library.

For necessary defines, see the **WOLFSSL_STM32XX** defines in settings.h. For example, the WOLFSSL_STM32F2 define enables STM32F2 hardware crypto and RNG support by default. The defines for enabling these individually are **`STM32_CRYPTO`** (for hardware crypto support) and **`STM32_RNG`** (for hardware RNG support).

Documentation and downloads for the STM32 Standard Peripheral Libraries can be found in the page:
https://www.st.com/en/embedded-software/stm32-standard-peripheral-libraries.html?querycriteria=productId=LN1939

4.4.3 Cavium NITROX

wolfSSL has support for Cavium NITROX (http://www.cavium.com/processor_security.html). To enable Cavium NITROX support when building wolfSSL use the following configure option:

```
./configure --with-cavium=/home/user/cavium/software
```

Where the "**--with-cavium=**" option is pointing to your licensed cavium/software directory. Since Cavium doesn't build a library wolfSSL pulls in the cavium_common.o file which gives a libtool warning about the portability of this. Also, if you're using the github source tree you'll need to remove the -Wredundant-decls warning from the generated Makefile because the cavium headers don't conform to this warning.

Currently wolfSSL supports Cavium RNG, AES, 3DES, RC4, HMAC, and RSA directly at the crypto layer. Support at the SSL level is partial and currently just does AES, 3DES, and RC4. RSA and HMAC are slower until the Cavium calls can be utilized in non-blocking mode. The example client turns on cavium support as does the crypto test and benchmark. Please see the **HAVE_CAVIUM** define.

4.4.4 Intel QuickAssist Technology (QAT)

wolfSSL has support for using hardware acceleration available through the Intel QAT drivers, and for the Intel QuickAssist 8970 hardware. Currently, the highest version of these drivers that are supported by wolfSSL for use with hardware accelerated cryptography is version 1.7. This version of those drivers provides support for asynchronous cryptography, RSA key generation, and SHA-3.

This hardware acceleration can be enabled by passing optional arguments to the wolfSSL configure script, or by adding specific "defines" to the **user_settings.h** file that would be used <u>and</u> by modifying the build configurations.

Option to pass to **configure**:

```
./configure --with-intelqa=PATH #where PATH is the path to the
Intel QAT driver directory
```

Options to define in **user_settings.h**:

```
#define HAVE_INTEL_QA
#define DO_CRYPTO
#define USER_SPACE
```

Build configurations to modify:

1. Add the QAT driver location to the include path
2. Add the QAT library to the linker

4.4.5 Xilinx SoCs and FPGAs

wolfSSL supports Xilinx SoCs and FPGAs. The wolfSSL embedded SSL/TLS library can be used with FPGAs which use the MicroBlaze CPU and/or Zynq and Zynq UltraScale+ SoCs. Improved performance speeds with using the hardware crpyto can be seen. Increasing AES-GCM, RSA, and SHA3 operations performance. In addition to the performance gained a user also gets the additional security the hardware provides while executing the algorithms.

The hardware acceleration and improvements can be enabled by passing optional arguments to the wolfSSL configure script, or by adding specific "defines" to the **user_settings.h** file that would be used <u>and</u> by modifying the build configurations.

Option to pass to **configure**:

```
./configure --enable-xilinx # may also be combined with --
enable-afalg=xilinx to tell wolfSSL to use Xilinx for AF_ALG.
```

Options to define in **user_settings.h**:

```
#define WOLFSSL_XILINX
#define WOLFSSL_XILINX_CRYPT
```

4.5 SSL Inspection (Sniffer)

Beginning with the wolfSSL 1.5.0 release, wolfSSL has included a build option allowing it to be built with SSL Sniffer (SSL Inspection) functionality. This means that you can collect SSL traffic packets and with the correct key file, are able to decrypt them as well. The ability to “inspect” SSL traffic can be useful for several reasons, some of which include:

- Analyzing Network Problems
- Detecting network misuse by internal and external users
- Monitoring network usage and data in motion
- Debugging client/server communications

To enable sniffer support, build wolfSSL with the **--enable-sniffer** option on *nix or use the **vcproj** files on Windows. You will need to have **pcap** installed on *nix or **WinPcap** on Windows. The main sniffer functions which can be found in *sniffer.h* are listed below with a short description of each:

ssl_SetPrivateKey - Sets the private key for a specific server and port.
ssl_SetNamedPrivateKey - Sets the private key for a specific server, port and domain name.
ssl_DecodePacket - Passes in a TCP/IP packet for decoding.
ssl_Trace - Enables / Disables debug tracing to the traceFile.
ssl_InitSniffer - Initialize the overall sniffer.
ssl_FreeSniffer - Free the overall sniffer.
ssl_EnableRecovery - Enables option to attempt to pick up decoding of SSL traffic in the case of lost packets.
ssl_GetSessionStats - Obtains memory usage for the sniffer sessions.

To look at wolfSSL's sniffer support and see a complete example, please see the "**snifftest**" app in the "sslSniffer/sslSnifferTest" folder from the wolfSSL download.

Keep in mind that because the encryption keys are setup in the SSL Handshake, the handshake needs to be decoded by the sniffer in order for future application data to be decoded. For example, if you are using "snifftest" with the wolfSSL example echoserver and echoclient, the snifftest application must be started before the handshake begins between the server and client.

The sniffer can only decode streams encrypted with the following algorithms: AES-CBC, DES3-CBC, ARC4, HC-128, RABBIT, Camellia-CBC, and IDEA. If ECDHE or DHE key agreement is used the stream cannot be sniffed; only RSA key-exchange is supported.

4.6 Compression

wolfSSL supports data compression with the **zlib** library. The ./configure build system detects the presence of this library, but 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 *wolfSSL_set_compression()* before SSL connecting or accepting. Both the client and server must have compression turned on in order for compression to be used.

Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

4.7 Pre-Shared Keys

wolfSSL has support for these ciphers with static pre-shared keys:

TLS_PSK_WITH_AES_256_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA256
TLS_PSK_WITH_AES_256_CBC_SHA384
TLS_PSK_WITH_AES_128_CBC_SHA
TLS_PSK_WITH_NULL_SHA256
TLS_PSK_WITH_NULL_SHA384
TLS_PSK_WITH_NULL_SHA
TLS_PSK_WITH_AES_128_GCM_SHA256
TLS_PSK_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_CCM
TLS_PSK_WITH_AES_256_CCM

TLS_PSK_WITH_AES_128_CCM_8
TLS_PSK_WITH_AES_256_CCM_8
TLS_PSK_WITH_CHACHA20_POLY1305

These suites are built into wolfSSL with **WOLFSSL_STATIC_PSK** on, all PSK suites can be turned off at build time with the constant **NO_PSK**. To only use these ciphers at runtime use the function **wolfSSL_CTX_set_cipher_list**() with the desired ciphersuite.

wolfSSL has support for ephemeral key PSK suites:

ECDHE-PSK-AES128-CBC-SHA256
ECDHE-PSK-NULL-SHA256
ECDHE-PSK-CHACHA20-POLY1305
DHE-PSK-CHACHA20-POLY1305
DHE-PSK-AES256-GCM-SHA384
DHE-PSK-AES128-GCM-SHA256
DHE-PSK-AES256-CBC-SHA384
DHE-PSK-AES128-CBC-SHA256
DHE-PSK-AES128-CBC-SHA256

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

On the server side two additional calls are required:

wolfSSL_CTX_set_psk_server_callback()
wolfSSL_CTX_use_psk_identity_hint()

The server stores its identity hint to help the client with the 2nd call, in our server example that's "wolfssl server". An example server psk callback can also be found in my_psk_server_cb() in wolfssl/test.h.

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

4.8 Client Authentication

Client authentication is a feature which enables the server to authenticate clients by requesting that the clients send a certificate to the server for authentication when they connect. Client authentication requires an X.509 client certificate from a CA (or self-signed if generated by you or someone other than a CA).

By default, wolfSSL validates all certificates that it receives - this includes both client and server. To set up client authentication, the server must load the list of trusted CA certificates to be used to verify the client certificate against:

```
wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);
```

To turn on client verification and control its behavior, the wolfSSL_CTX_set_verify() function is used. In the following example, **SSL_VERIFY_PEER** turns on a certificate request from the server to the client. **SSL_VERIFY_FAIL_IF_NO_PEER_CERT** instructs the server to fail if the client does not present a certificate to validate on the server side. Other options to wolfSSL_CTX_set_verify() include SSL_VERIFY_NONE and SSL_VERIFY_CLIENT_ONCE.

```
wolfSSL_CTX_set_verify(ctx,SSL_VERIFY_PEER | ((usePskPlus)?
                        SSL_VERIFY_FAIL_EXCEPT_PSK :
                        SSL_VERIFY_FAIL_IF_NO_PEER_CERT),0);
```

An example of client authentication can be found in the example server (server.c) included in the wolfSSL download (/examples/server/server.c).

4.9 Server Name Indication

SNI is useful when a server hosts multiple ‘virtual’ servers at a single underlying network address. It may be desirable for clients to provide the name of the server which it is contacting. To enable SNI with wolfSSL you can simply do:

```
./configure --enable-sni
```

Using SNI on the client side requires an additional function call, which should be one of the following functions:

```
wolfSSL_CTX_UseSNI()
wolfSSL_UseSNI()
```

wolfSSL_CTX_UseSNI() is most recommended when the client contacts the same server multiple times. Setting the SNI extension at the context level will enable the SNI usage in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseSNI() will enable SNI usage for one SSL object only, so it is recommended to use this function when the server name changes between sessions.

On the server side one of the same function calls is required. Since the wolfSSL server doesn't host multiple 'virtual' servers, the SNI usage is useful when the termination of the connection is desired in the case of SNI mismatch. In this scenario, wolfSSL_CTX_UseSNI() will be more efficient, as the server will set it only once per context creating all subsequent SSL objects with SNI from that same context.

4.10 Handshake Modifications

4.10.1 Grouping Handshake Messages

wolfSSL has the ability to group handshake messages if the user desires. This can be done at the context level with:

```
wolfSSL_CTX_set_group_messages(ctx);
```

or at the SSL object level with:

```
wolfSSL_set_group_messages(ssl);
```

4.11 Truncated HMAC

Currently defined TLS cipher suites use the HMAC to authenticate record-layer communications. In TLS, the entire output of the hash function is used as the MAC tag. However, it may be desirable in constrained environments to save bandwidth by truncating the output of the hash function to 80 bits when forming MAC tags. To enable the usage of Truncated HMAC at wolfSSL you can simply do:

```
./configure --enable-truncatedhmac
```

Using Truncated HMAC on the client side requires an additional function call, which should be one of the following functions:

```
wolfSSL_CTX_UseTruncatedHMAC();
wolfSSL_UseTruncatedHMAC();
```

wolfSSL_CTX_UseTruncatedHMAC() is most recommended when the client would like to enable Truncated HMAC for all sessions. Setting the Truncated HMAC extension at context level will enable it in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseTruncatedHMAC() will enable it for one SSL object only, so it's recommended to use this function when there is no need for Truncated HMAC on all sessions.

On the server side no call is required. The server will automatically attend to the client's request for Truncated HMAC.

All TLS extensions can also be enabled with:

```
./configure --enable-tlsx
```

4.12 User Crypto Module

User Crypto Module allows for a user to plug in custom crypto that they want used during supported operations (Currently RSA operations are supported). An example of a module is located in the directory root_wolfssl/wolfcrypt/user-crypto/ using IPP libraries. Examples of the configure option when building wolfSSL to use a crypto module is as follows :

```
./configure --with-user-crypto
or
./configure --with-user-crypto=/dir/to
```

When creating a user crypto module that performs RSA operations, it is mandatory that there is a header file for RSA called user_rsa.h. For all user crypto operations it is mandatory that the users library be called libusercrypto. These are the names that wolfSSL autoconf tools will be looking for when linking and using a user crypto module. In the example provided with wolfSSL, the header file user_rsa.h can be found in the directory wolfcrypt/user-crypto/include/ and the library once created is located in the directory wolfcrypt/user-crypto/lib/ . For a list of required API look at the header file provided.

To build the example, after having installed IPP libraries, the following commands from the root wolfSSL directory should be ran.

```
cd wolfcrypt/user-crypto/
./autogen.sh
./configure
make
sudo make install
```

The included example in wolfSSL requires the use of IPP, which will need to be installed before the project can be built. Though even if not having IPP libraries to build the example it is intended to provide users with an example of file name choice and API interface. Once having made and installed both the library libusercrypto and header files, making wolfSSL use the crypto module does not require any extra steps. Simply using the configure flag --with-user-crypto will map all function calls from the typical wolfSSL crypto to the user crypto module.

Memory allocations, if using wolfSSL's XMALLOC, should be tagged with DYNAMIC_TYPE_USER_CRYPTO. Allowing for analyzing memory allocations used by the module.

User crypto modules **cannot** be used in conjunction with the wolfSSL configure options fast-rsa and/or fips. Fips requires that specific, certified code be used and fast-rsa makes use of the example user crypto module to perform RSA operations.

4.13 Timing-Resistance in wolfSSL

wolfSSL provides the function “ConstantCompare” which guarantees constant time when doing comparison operations that could potentially leak timing information. This API is used at both the TLS and crypto level in wolfSSL to deter against timing based, side-channel attacks.

The wolfSSL ECC implementation has the define ECC_TIMING_RESISTANT to enable timing-resistance in the ECC algorithm. Similarly the define TFM_TIMING_RESISTANT is provided in the fast math libraries for RSA algorithm timing-resistance. The function exptmod uses the timing resistant Montgomery ladder.

See also: --enable-harden

Chapter 5: Portability

5.1 Abstraction Layers

5.1.1 C Standard Library Abstraction Layer

wolfSSL (formerly CyaSSL) can be built without the C standard library to provide a higher level of portability and flexibility to developers. The user will have to map the functions they wish to use instead of the C standard ones.

5.1.1.1 Memory Use

Most C programs use *malloc()* and *free()* for dynamic memory allocation. wolfSSL uses **XMALLOC()** and **XFREE()** instead. By default, these point to the C runtime versions. By defining XMALLOC_USER, the user can provide their own hooks. Each memory function takes two additional arguments over the standard ones, a heap hint, and an allocation type. The user is free to ignore these or use them in any way they like. You can find the wolfSSL memory functions in **wolfssl/wolfcrypt/types.h**.

wolfSSL also provides the ability to register memory override functions at runtime instead of compile time. **wolfssl/wolfcrypt/memory.h** is the header for this functionality and the user can call the following function to set up the memory functions:

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb  malloc_function,
                          wolfSSL_Free_cb    free_function,
                          wolfSSL_Realloc_cb realloc_function);
```

See the header **wolfssl/wolfcrypt/memory.h** for the callback prototypes and **memory.c** for the implementation.

5.1.1.2 string.h

wolfSSL uses several functions that behave like string.h's *memcpy()*, *memset()*, and *memcmp()* amongst others. They are abstracted to **XMEMCPY()**, **XMEMSET()**, and **XMEMCMP()** respectively. And by default, they point to the C standard library versions. Defining STRING_USER allows the user to provide their own hooks in types.h. For example, by default **XMEMCPY()** is:

```
#define XMEMCPY(d,s,l)    memcpy((d),(s),(l))
```

After defining STRING_USER you could do:

```
#define XMEMCPY(d,s,l)    my_memcpy((d),(s),(l))
```

Or if you prefer to avoid macros:

```
external void* my_memcpy(void* d, const void* s, size_t n);
```

to set wolfSSL's abstraction layer to point to your version my_memcpy().

5.1.1.3 math.h

wolfSSL uses two functions that behave like math.h's *pow()* and *log()*. They are only required by Diffie-Hellman, so if you exclude DH from the build, then you don't have to provide your own. They are abstracted to **XPOW()** and **XLOG()** and found in **wolfcrypt/src/dh.c**.

5.1.1.4 File System Use

By default, wolfSSL uses the system's file system for the purpose of loading keys and certificates. This can be turned off by defining NO_FILESYSTEM, see item V. If instead, you'd like to use a file system but not the system one, you can use the **XFILE()** layer in **ssl.c** to point the file system calls to the ones you'd like to use. See the example provided by the MICRIUM define.

5.1.2 Custom Input/Output Abstraction Layer

wolfSSL provides a custom I/O abstraction layer for those who wish to have higher control over I/O of their SSL connection or run SSL on top of a different transport medium other than TCP/IP.

The user will need to define two functions:

1. The network Send function
2. The network Receive function

These two functions are prototyped by **CallbackIOSend** and **CallbackIORecv** in **ssl.h**:

```
typedef int (*CallbackIORecv)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

The user needs to register these functions per WOLFSSL_CTX with **wolfSSL_SetIOSend()** and **wolfSSL_SetIORecv()**. For example, in the default case, CBIORecv() and CBIOSend() are registered at the bottom of **io.c**:

```
void wolfSSL_SetIORecv(WOLFSSL_CTX *ctx, CallbackIORecv CBIORecv)
{
     ctx->CBIORecv = CBIORecv;
}

void wolfSSL_SetIOSend(WOLFSSL_CTX *ctx, CallbackIOSend CBIOSend)
{
    ctx->CBIOSend = CBIOSend;
}
```

The user can set a context per WOLFSSL object (session) with **wolfSSL_SetIOWriteCtx()** and **wolfSSL_SetIOReadCtx()**, as demonstrated at the bottom of **io.c**. For example, if the user is using memory buffers, the context may be a pointer to a structure describing where and how to access the memory buffers. The default case, with no user overrides, registers the socket as the context.

The CBIORecv and CBIOSend function pointers can be pointed to your custom I/O functions. The default Send() and Receive() functions, **EmbedSend()** and

EmbedReceive(), located in **io.c**, can be used as templates and guides.

WOLFSSL_USER_IO can be defined to remove the automatic setting of the default I/O functions EmbedSend() and EmbedReceive().

5.1.3 Operating System Abstraction Layer

The wolfSSL OS abstraction layer helps facilitate easier porting of wolfSSL to a user's operating system. The **wolfssl/wolfcrypt/settings.h** file contains settings which end up triggering the OS layer.

OS-specific defines are located in **wolfssl/wolfcrypt/types.h** for wolfCrypt and **wolfssl/internal.h** for wolfSSL.

5.2 Supported Operating Systems

One factor which defines wolfSSL is its ability to be easily ported to new platforms. As such, wolfSSL has support for a long list of operating systems out-of-the-box. Currently-supported operating systems include:

Win32/64, Linux, Mac OS X, Solaris, ThreadX, VxWorks, FreeBSD, NetBSD, OpenBSD, embedded Linux, Yocto Linux, OpenEmbedded, WinCE, Haiku, OpenWRT, iPhone (iOS), Android, Nintendo Wii and Gamecube through DevKitPro, QNX, MontaVista, NonStop, TRON/ITRON/µITRON, Micrium's µC/OS-III, FreeRTOS, SafeRTOS, NXP/Freescale MQX, Nucleus, TinyOS, HP/UX, AIX, ARC MQX, TI-RTOS, uTasker, embOS, INtime, Mbed, µT-Kernel, RIOT, CMSIS-RTOS, FROSTED, Green Hills INTEGRITY, Keil RTX, TOPPERS, PetaLinux, Apache Mynewt

5.3 Supported Chipmakers

wolfSSL has support for chipsets including ARM, Intel, Motorola, mbed, Freescale, Microchip (PIC32), STMicro (STM32F2/F4), NXP, Analog Devices, Texas Instruments, AMD and more.

5.4 C# Wrapper

wolfSSL has limited support for use in C#. A Visual Studio project containing the port can be found in the directory “root_wolfSSL/wrapper/CSharp/”. After opening the Visual Studio project set the “Active solution configuration” and “Active solution platform” by clicking on BUILD->Configuration Manager… The supported “Active solution configuration”s are DLL Debug and DLL Release. The supported platforms are Win32 and x64.

Once having set the solution and platform the preprocessor flag **HAVE_CSHARP** will need to be added. This turns on the options used by the C# wrapper and used by the examples included.

To then build simply select build solution. This creates the wolfssl.dll, wolfSSL_CSharp.dll and examples. Examples can be ran by targeting them as an entry point and then running debug in Visual Studio.

Adding the created C# wrapper to C# projects can be done a couple of ways. One way is to install the created wolfssl.dll and wolfSSL_CSharp.dll into the directory C:/Windows/System/. This will allow projects that have

```
using wolfSSL.CSharp

public some_class {

      public static main(){
            wolfssl.Init()
            ...
      }
...
```

to make calls to the wolfSSL C# wrapper. Another way is to create a Visual Studio project and have it reference the bundled C# wrapper solution in wolfSSL.

Chapter 6: Callbacks

6.1 HandShake Callback

wolfSSL (formerly CyaSSL) has an extension that allows a HandShake Callback to be set for connect or accept. This can be useful in embedded systems for debugging support when another debugger isn't available and sniffing is impractical. To use wolfSSL HandShake Callbacks, use the extended functions, ***wolfSSL_connect_ex()*** and ***wolfSSL_accept_ex()***:

```
int wolfSSL_connect_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                       Timeval)
int wolfSSL_accept_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                      Timeval)
```

HandShakeCallBack is defined as:

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

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

```
typedef struct handShakeInfo_st {
    char   cipherName[MAX_CIPHERNAME_SZ + 1];  /*negotiated name */
    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.

6.2 Timeout Callback

The same extensions used with wolfSSL Handshake Callbacks can be used for wolfSSL Timeout Callbacks as well. These extensions can be called with either, both, or neither callbacks (Handshake and/or Timeout). *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 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 wolfSSL 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.

6.3 User Atomic Record Layer Processing

wolfSSL provides Atomic Record Processing callbacks for users who wish to have more control over MAC/encrypt and decrypt/verify functionality during the SSL/TLS connection.

The user will need to define 2 functions:

1. MAC/encrypt callback function
2. Decrypt/verify callback function

These two functions are prototyped by **CallbackMacEncrypt** and **CallbackDecryptVerify** in **ssl.h**:

```
typedef int (*CallbackMacEncrypt)(WOLFSSL* ssl,
    unsigned char* macOut,const unsigned char* macIn,
    unsigned int macInSz,int macContent, int macVerify,
    unsigned char* encOut, const unsigned char* encIn,
    unsigned int encSz,void* ctx);

typedef int (*CallbackDecryptVerify)(WOLFSSL* ssl,
    unsigned char* decOut, const unsigned char* decIn,
    unsigned int decSz, int content, int verify,
    unsigned int* padSz, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with **wolfSSL_CTX_SetMacEncryptCb()** and **wolfSSL_CTX_SetDecryptVerifyCb()**.

The user can set a context per WOLFSSL object (session) with **wolfSSL_SetMacEncryptCtx()** and **wolfSSL_SetDecryptVerifyCtx()**. This context may be a pointer to any user-specified context, which will then in turn be passed back to the MAC/encrypt and decrypt/verify callbacks through the “void* ctx” parameter.

1. Example callbacks can be found in wolfssl/test.h, under myMacEncryptCb() and myDecryptVerifyCb(). Usage can be seen in the wolfSSL example client (examples/client/client.c), when using the “-U” command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the “--enable-atomicuser” configure option, or by defining the **ATOMIC_USER** preprocessor flag.

6.4 Public Key Callbacks

wolfSSL provides Public Key callbacks for users who wish to have more control over

ECC sign/verify functionality as well as RSA sign/verify and encrypt/decrypt functionality during the SSL/TLS connection.

The user can optionally define 7 functions:

1. ECC sign callback
2. ECC verify callback
3. ECC shared secret callback
4. RSA sign callback
5. RSA verify callback
6. RSA encrypt callback
7. RSA decrypt callback

These two functions are prototyped by **CallbackEccSign**, **CallbackEccVerify, CallbackEccSharedSecret**, **CallbackRsaSign**, **CallbackRsaVerify**, **CallbackRsaEnc**, and **CallbackRsaDec** in **ssl.h**:

```
typedef int (*CallbackEccSign)(WOLFSSL* ssl, const unsigned
               char* in, unsigned int inSz, unsigned char* out,
               unsigned int* outSz, const unsigned char* keyDer,
               unsigned int keySz, void* ctx);

typedef int (*CallbackEccVerify)(WOLFSSL* ssl,
          const unsigned char* sig, unsigned int sigSz,
          const unsigned char* hash, unsigned int hashSz,
          const unsigned char* keyDer, unsigned int keySz,
          int* result, void* ctx);

typedef int (*CallbackEccSharedSecret)(WOLFSSL* ssl,
          struct ecc_key* otherKey,
          unsigned char* pubKeyDer, unsigned int* pubKeySz,
          unsigned char* out, unsigned int* outlen,
          int side, void* ctx);

typedef int (*CallbackRsaSign)(WOLFSSL* ssl,
          const unsigned char* in, unsigned int inSz,
          unsigned char* out, unsigned int* outSz,
          const unsigned char* keyDer, unsigned int keySz,
          void* ctx);

typedef int (*CallbackRsaVerify)(WOLFSSL* ssl,
```

```
          unsigned char* sig, unsigned int sigSz,
          unsigned char** out, const unsigned char* keyDer,
          unsigned int keySz, void* ctx);

typedef int (*CallbackRsaEnc)(WOLFSSL* ssl,
          const unsigned char* in, unsigned int inSz,
          Unsigned char* out, unsigned int* outSz,
          const unsigned char* keyDer,
          unsigned int keySz, void* ctx);

typedef int (*CallbackRsaDec)(WOLFSSL* ssl, unsigned char* in,
          unsigned int inSz, unsigned char** out,
          const unsigned char* keyDer, unsigned int keySz,
          void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with **wolfSSL_CTX_SetEccSignCb()**, **wolfSSL_CTX_SetEccVerifyCb(), wolfSSL_CTX_SetEccSharedSecretCb(), wolfSSL_CTX_SetRsaSignCb()**, **wolfSSL_CTX_SetRsaVerifyCb()**, **wolfSSL_CTX_SetRsaEncCb()**, and **wolfSSL_CTX_SetRsaDecCb()**.

The user can set a context per WOLFSSL object (session) with **wolfSSL_SetEccSignCtx()**, **wolfSSL_SetEccVerifyCtx(), wolfSSL_SetEccSharedSecretCtx(), wolfSSL_SetRsaSignCtx()**, **wolfSSL_SetRsaVerifyCtx()**, **wolfSSL_SetRsaEncCtx()**, and **wolfSSL_SetRsaDecCtx()**. These contexts may be pointers to any user-specified context, which will then in turn be passed back to the respective public key callback through the “void* ctx” parameter.

Example callbacks can be found in wolfssl/test.h, under myEccSign(), myEccVerify(), myEccSharedSecret(), myRsaSign(), myRsaVerify(), myRsaEnc(), and myRsaDec(). Usage can be seen in the wolfSSL example client (examples/client/client.c), when using the “-P” command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the **--enable-pkcallbacks** configure option, or by defining the **HAVE_PK_CALLBACKS** preprocessor flag.

Chapter 7: Keys and Certificates

For an introduction to X.509 certificates, as well as how they are used in SSL and TLS, please see Appendix A.

7.1 Supported Formats and Sizes

wolfSSL (formerly CyaSSL) has support for **PEM**, and **DER** formats for certificates and keys, as well as PKCS#8 private keys (with PKCS#5 or PKCS#12 encryption).

PEM, or "Privacy Enhanced Mail" is the most common format that certificates are issued in by certificate authorities. PEM files are Base64 encoded ASCII files which can include multiple server certificates, intermediate certificates, and private keys, and usually have a **.pem**, **.crt**, **.cer**, or **.key** file extension. Certificates inside PEM files are wrapped in the "`-----BEGIN CERTIFICATE-----`" and "`-----END CERTIFICATE-----`" statements.

DER, or "Distinguished Encoding Rules", is a binary format of a certificate. DER file extensions can include **.der** and **.cer**, and cannot be viewed with a text editor.

An X.509 certificate is encoded using ASN.1 format. The DER format is the ASN.1 encoding. The PEM format is Base64 encoded and wrapped with a human readable header and footer. TLS send certificates in DER format.

7.2 Certificate Loading

Certificates are normally loaded using the file system (although loading from memory buffers is supported as well - see **Section 7.5**).

7.2.1 Loading CA Certificates

CA certificate files can be loaded using the wolfSSL_CTX_load_verify_locations() function:

```
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX *ctx,
                                      const char *CAfile,
                                      const char *CApath);*
```

CA loading can also parse multiple CA certificates per file using the above function by passing in a **CAfile** in PEM format with as many certs as possible. This makes initialization easier, and is useful when a client needs to load several root CAs at startup. This makes wolfSSL easier to port into tools that expect to be able to use a single file for CAs.

NOTICE: If you have to load a chain of Roots and Intermediate certificates you must load them in the order of trust. Load ROOT CA first followed by Intermediate 1 followed by Intermediate 2 and so on. You may call wolfSSL_CTX_load_verify_locations for each cert to be loaded or just once with a file containing the certs in order (Root at the top of the file and certs ordered by the chain of trust)

7.2.2 Loading Client or Server Certificates

Loading single client or server certificates can be done with the wolfSSL_CTX_use_certificate_file() function. If this function is used with a certificate chain, only the actual, or "bottom" certificate will be sent.

```
int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     int type);
```

CAfile is the CA certificate file, and **type** is the format of the certificate - such as SSL_FILETYPE_PEM.

The server and client can send certificate chains using the *wolfSSL_CTX_use_certificate_chain_file()* function. The certificate chain file must be in PEM format and must be sorted starting with the subject's certificate (the actual client or server cert), followed by any intermediate certificates and ending (optionally) at the root "top" CA. The example server (/examples/server/server.c) uses this functionality.

NOTICE: This is the exact reverse of the order necessary when loading a certificate chain for verification! Your file contents in this scenario would be Entity cert at the top of the file followed by the next cert up the chain and so on with Root CA at the bottom of the file.

```
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                           const char *file);
```

7.2.3 Loading Private Keys

Server private keys can be loaded using the wolfSSL_CTX_use_PrivateKey_file() function.

```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                        const char *keyFile, int type);
```

keyFile is the private key file, and **type** is the format of the private key (e.g. SSL_FILETYPE_PEM).

7.2.4 Loading Trusted Peer Certificates

Loading a trusted peer certificate to use can be done with wolfSSL_CTX_trust_peer_cert().

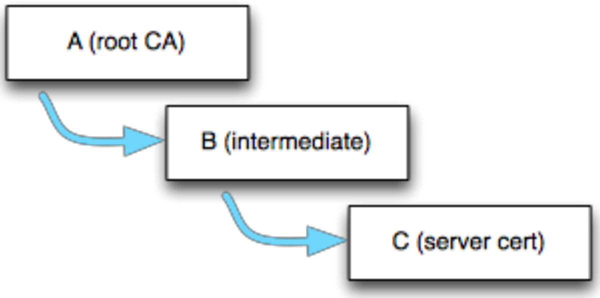
```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,
                    const char *trustCert, int type);
```

trustCert is the certificate file to load, and **type** is the format of the private key (i.e. SSL_FILETYPE_PEM).

7.3 Certificate Chain Verification

wolfSSL requires that only the top or “root” certificate in a chain to be loaded as a trusted certificate in order to verify a certificate chain. This means that if you have a certificate chain (A -> B -> C), where C is signed by B, and B is signed by A, wolfSSL only requires that certificate A be loaded as a trusted certificate in order to verify the entire chain (A->B->C).

For example, if a server certificate chain looks like:



The wolfSSL client should already have at least the root cert (A) loaded as a trusted root (with wolfSSL_CTX_load_verify_locations()). When the client receives the server cert chain, it uses the signature of A to verify B, and if B has not been previously loaded into wolfSSL as a trusted root, B gets stored in wolfSSL's internal cert chain (wolfSSL just stores what is necessary to verify a certificate: common name hash, public key and key type, etc.). If B is valid, then it is used to verify C.

Following this model, as long as root cert "A" has been loaded as a trusted root into the wolfSSL server, the server certificate chain will still be able to be verified if the server sends (A->B->C), or (B->C). If the server just sends (C), and not the intermediate certificate, the chain will not be able to be verified unless the wolfSSL client has already loaded B as a trusted root.

7.4 Domain Name Check for Server Certificates

wolfSSL 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. wolfSSL checks that the date of the certificate is in range, verifies the signature, and additionally verifies the domain if you call:

```
wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn)
```

before calling wolf*SSL_connect()*. wolfSSL will match the X.509 issuer name of peer's server certificate against **dn** (the expected domain name). If the names match wolf*SSL_connect()* will proceed normally, however if there is a name mismatch, wolf*SSL_connect()* will return a fatal error and wolf*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.

7.5 No File System and using Certificates

Normally a file system is used to load private keys, certificates, and CAs. Since wolfSSL is sometimes used in environments without a full file system 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 wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned
                                 char* in,long sz, int format);
int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx,
                                      const unsigned char* in,
                                      long sz, int format);
int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx,
                                      const unsigned char* in,
                                      long sz, int format);
int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx,
                                   const unsigned char* in,long sz);
int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx,
                                      const unsigned char* in,
                                      Long sz, int format);
```

Use these functions exactly like their counterparts that are named "*_*file*" instead of "*_*buffer*". And instead of providing a filename provide a memory buffer. See API documentation for usage details.

7.5.1 Test Certificate and Key Buffers

wolfSSL has come bundled with test certificate and key files in the past. Now it also comes bundled with test certificate and key buffers for use in environments with no filesystem available. These buffers are available in certs_test.h when defining one or more of **USE_CERT_BUFFERS_1024**, **USE_CERT_BUFFERS_2048**, or **USE_CERT_BUFFERS_256**.

7.6 Serial Number Retrieval

The serial number of an X.509 certificate can be extracted from wolfSSL using the following function. The serial number can be of any length.

```
int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509,
                       unsigned char* buffer, int* inOutSz)
```

buffer will be written to with at most ***inOutSz** bytes on input. After the call, if successful (return of 0), ***inOutSz** will hold the actual number of bytes written to **buffer**. A full example is included wolfssl/test.h.

7.7 RSA Key Generation

wolfSSL 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 **WOLFSSL_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 wolfcrypt/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, wolfSSL 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.

7.7.1 RSA Key Generation Notes

Although an RSA private key contains the public key as well, wolfSSL provides functionality for exporting an RSA public key from an RSA key structure through the `wc_RsaKeyToPublicDer` function. This function requires that `WOLFSSL_KEY_GEN` be defined when building the library. The signature of this function is shown below:

```
int wc_RsaKeyToPublicDer(RsaKey* key,
     byte* output,
     word32 inLen);
```

7.8 Certificate Generation

wolfSSL supports X.509 v3 certificate generation. Certificate generation is off by default but can be turned on during the ./configure process with:

--enable-certgen

or by defining **WOLFSSL_CERT_GEN** in Windows or non-standard 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 *wolfssl/ wolfcrypt/asn_public.h* named *Cert*:

```
/* for user to fill for certificate generation */
typedef struct Cert {
    int      version;                  /* x509 version  */
    byte     serial[CTC_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 */
    int      isCA;                     /*is this going to be a CA*/
    ...
} Cert;
```

Where CertName looks like:

```
typedef struct CertName {
     char country[CTC_NAME_SIZE];
     char countryEnc;
     char state[CTC_NAME_SIZE];
     char stateEnc;
     char locality[CTC_NAME_SIZE];
     char localityEnc;
     char sur[CTC_NAME_SIZE];
     char surEnc;
     char org[CTC_NAME_SIZE];
     char orgEnc;
     char unit[CTC_NAME_SIZE];
     char unitEnc;
     char commonName[CTC_NAME_SIZE];
     char commonNameEnc;
     char email[CTC_NAME_SIZE];  /* !!!! email has to be last!!!! */
} 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 **CTC_SHAwRSA**, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

```
CTC_SHAwDSA
CTC_MD2wRSA
CTC_MD5wRSA
CTC_SHAwRSA
CTC_SHAwECDSA
CTC_SHA256wRSA
CTC_SHA256wECDSA
CTC_SHA384wRSA
CTC_SHA384wECDSA
CTC_SHA512wRSA
CTC_SHA512wECDSA
```

Now the user can initialize the subject information like this example from **wolfcrypt/test/test.c**:

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

Then, a self-signed 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 = MakeSelfCert(&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* pem;

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

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

If you wish to create a CA signed certificate then a couple of steps are required. After filling in the subject information as before, you'll need to set the issuer information from the CA certificate. This can be done with *SetIssuer()* like this:

```
ret = SetIssuer(&myCert, "ca-cert.pem");
if (ret < 0)
      /* ret contains error */;
```

Then you'll need to perform the two-step process of creating the certificate and then signing it (*MakeSelfCert()* does these both in one step). You'll need the private keys from both the issuer (**caKey**) and the subject (**key**). Please see the example in **test.c** for complete usage.

```
byte derCert[4096];

int certSz = MakeCert(&myCert, derCert, sizeof(derCert), &key, NULL,
&rng);

if (certSz < 0);
   /*certSz contains the error*/;

certSz = SignCert(myCert.bodySz, myCert.sigType, derCert,

                      sizeof(derCert), &caKey, NULL, &rng);

if (certSz < 0);
   /*certSz contains the error*/;
```

The buffer *derCert* now contains a DER format of the CA signed certificate. If you need a PEM format of the certificate please see the self signed example above. Note that *MakeCert()* and *SignCert()* provide function parameters for either an RSA or ECC key to be used. The above example uses an RSA key and passes NULL for the ECC key parameter.

7.9 Certificate Signing Request (CSR) Generation

wolfSSL supports X.509 v3 certificate signing request (CSR) generation. CSR generation is off by default but can be turned on during the ./configure process with:

--enable-certreq --enable-certgen

or by defining **WOLFSSL_CERT_GEN** and **WOLFSSL_CERT_REQ** in Windows or non-standard environments.

Before a CSR can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from *wolfssl/wolfcrypt/asn_public.h* named *Cert*:

For details on the Cert and CertName structures please reference section “7.8 Certificate Generation” above.

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

```
Cert request;
InitCert(&request);
```

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 **CTC_SHAwRSA**, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

```
CTC_SHAwDSA
CTC_MD2wRSA
CTC_MD5wRSA
CTC_SHAwRSA
CTC_SHAwECDSA
CTC_SHA256wRSA
CTC_SHA256wECDSA
CTC_SHA384wRSA
CTC_SHA384wECDSA
CTC_SHA512wRSA
CTC_SHA512wECDSA
```

Now the user can initialize the subject information like this example from **https://github.com/wolfSSL/wolfssl-examples/blob/master/certgen/csr_example.c**:

```
strncpy(req.subject.country, "US", CTC_NAME_SIZE);
strncpy(req.subject.state, "OR", CTC_NAME_SIZE);
strncpy(req.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(req.subject.org, "wolfSSL", CTC_NAME_SIZE);
strncpy(req.subject.unit, "Development", CTC_NAME_SIZE);
strncpy(req.subject.commonName, "www.wolfssl.com",
CTC_NAME_SIZE);
strncpy(req.subject.email, "info@wolfssl.com", CTC_NAME_SIZE);
```

Then, a valid signed CSR can be generated using the variable key from the above key generation example (of course any valid ECC/RSA key or RNG can be used):

```
byte der[4096]; /* Store request in der format once made */

ret = wc_MakeCertReq(&request, der, sizeof(der), NULL, &key);
/* check ret value for error handling, <= 0 indicates a failure */
```

Next you will want to sign your request making it valid, use the rng variable from the above key generation example. (of course any valid ECC/RSA key or RNG can be used)

```
derSz = ret;

req.sigType = CTC_SHA256wECDSA;
ret = wc_SignCert(request.bodySz, request.sigType, der, sizeof(der),
NULL, &key, &rng);
/* check ret value for error handling, <= 0 indicates a failure */
```

Lastly it is time to convert the CSR to PEM format for sending to a CA authority to use in issueing a certificate:

```
ret = wc_DerToPem(der, derSz, pem, sizeof(pem), CERTREQ_TYPE);
/* check ret value for error handling, <= 0 indicates a failure */
printf("%s", pem); /* or write to a file */
```

Limitations:
There are fields that are mandatory in a certificate that are excluded in a CSR. There are other fields in a CSR that are also deemed “optional” that are otherwise mandatory when in a certificate. Because of this the wolfSSL certificate parsing engine, which strictly checks all certificate fields AND considers all fields mandatory, does not support consuming a CSR at this time. Therefore while CSR generation AND certificate generation from scratch are supported, wolfSSL does not support certificate generation FROM a CSR. Passing in a CSR to the wolfSSL parsing engine will return a failure at this time. Check back for updates once we support consuming a CSR for use in certificate generation!

See also:
7.8 Certificate Generation

7.10 Convert to raw ECC key

With our recently added support for raw ECC key import comes the ability to convert an ECC key from PEM to DER. Use the following with the specified arguments to accomplish this:

```
EccKeyToDer(ecc_key*, byte* output, word32 inLen);
```

Example:

```
#define FOURK_BUF 4096
byte  der[FOURK_BUF];
ecc_key userB;

EccKeyToDer(&userB, der, FOURK_BUF);
```

Chapter 8: Debugging

8.1 Debugging and Logging

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

As of wolfSSL 2.0, logging callback functions may be registered at runtime to provide more flexibility with how logging is done. The logging callback can be registered with the following function:

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);
```

```
typedef void (*wolfSSL_Logging_cb)(const int logLevel,
                  const char *const logMessage);
```

The log levels can be found in **wolfssl/wolfcrypt/logging.h**, and the implementation is located in **logging.c**. By default, wolfSSL logs to *stderr* with *fprintf*.

8.2 Error Codes

wolfSSL tries to provide informative error messages in order to help with debugging.

Each wolfSSL_read() and wolfSSL_write() call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like read() and write(). In the event of an error you can use two calls to get more information about the error.

The function wolfSSL_get_error() will return the current error code. It takes the current WOLFSSL object, and wolfSSL_read() or wolfSSL_write() result value as an arguments and returns the corresponding error code.

```
int err = wolfSSL_get_error(ssl, result);
```

To get a more human-readable error code description, the wolfSSL_ERR_error_string() function can be used. It takes the return code from wolfSSL_get_error and a storage buffer as arguments, and places the corresponding error description into the storage buffer (**errorString** in the example below).

```
char errorString[80];
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non blocking sockets, you can test for errno EAGAIN/EWOULDBLOCK or more correctly you can test the specific error code for SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE.

For a list of wolfSSL and wolfCrypt error codes, please see Appendix C (Error Codes).

Chapter 9: Library Design

9.1 Library Headers

With the release of wolfSSL 2.0.0 RC3, library header files are now located in the following locations:

wolfSSL:	wolfssl/
wolfCrypt:	wolfssl/wolfcrypt/
wolfSSL OpenSSL Compatibility Layer:	wolfssl/openssl/

When using the OpenSSL Compatibility layer (see **Chapter 13**), the /wolfssl/openssl/ssl.h header is required to be included:

```
#include <wolfssl/openssl/ssl.h>
```

When using only the wolfSSL native API, only the /wolfssl/ssl.h header is required to be included:

```
#include <wolfssl/ssl.h>
```

9.2 Startup and Exit

All applications should call *wolfSSL_Init()* before using the library and call *wolfSSL_Cleanup()* 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.

9.3 Structure Usage

In addition to header file location changes, the release of wolfSSL 2.0.0 RC3 created a more visible distinction between the native wolfSSL API and the wolfSSL OpenSSL Compatibility Layer. With this distinction, the main SSL/TLS structures used by the native wolfSSL API have changed names. The new structures are as follows. The previous names are still used when using the OpenSSL Compatibility Layer (see **Chapter 13**).

```
WOLFSSL              (previously SSL)
```

```
WOLFSSL_CTX          (previously SSL_CTX)
WOLFSSL_METHOD       (previously SSL_METHOD)
WOLFSSL_SESSION      (previously SSL_SESSION)
WOLFSSL_X509         (previously X509)
WOLFSSL_X509_NAME    (previously X509_NAME)
WOLFSSL_X509_CHAIN   (previously X509_CHAIN)
```

9.4 Thread Safety

wolfSSL (formerly CyaSSL) is thread safe by design. Multiple threads can enter the library simultaneously without creating conflicts because wolfSSL avoids global data, static data, and the sharing of objects. The user must still take care to avoid potential problems in two areas.

1. A client may share an WOLFSSL object across multiple threads but access must be synchronized, i.e., trying to read/write at the same time from two different threads with the same SSL pointer is not supported.

   wolfSSL could take a more aggressive (constrictive) stance and lock out other users when a function is entered that cannot be shared but this level of granularity seems counter-intuitive. All users (even single threaded ones) will pay for the locking and multi-thread ones won't be able to re-enter the library even if they aren't sharing objects across threads. This penalty seems much too high and wolfSSL leaves the responsibility of synchronizing shared objects in the hands of the user.

2. Besides sharing WOLFSSL pointers, users must also take care to completely initialize an WOLFSSL_CTX before passing the structure to wolfSSL_new(). The same WOLFSSL_CTX can create multiple WOLFSSL structs but the WOLFSSL_CTX is only read during wolfSSL_new() creation and any future (or simultaneous changes) to the WOLFSSL_CTX will not be reflected once the WOLFSSL object is created.

   Again, multiple threads should synchronize writing access to a WOLFSSL_CTX and it is advised that a single thread initialize the WOLFSSL_CTX to avoid the synchronization and update problem described above.

9.5 Input and Output Buffers

wolfSSL now uses dynamic buffers for input and output. They default to 0 bytes and are controlled by the RECORD_SIZE define in **wolfssl/internal.h**. If an input record is received that is greater in size than the static buffer, then a dynamic buffer is temporarily used to handle the request and then freed. You can set the static buffer size up to the MAX_RECORD_SIZE which is 2^16 or 16,384.

If you prefer the previous way that wolfSSL operated, with 16Kb static buffers that will never need dynamic memory, you can still get that option by defining **LARGE_STATIC_BUFFERS**.

If dynamic buffers are used and the user requests a **wolfSSL_write**() that is bigger than the buffer size, then a dynamic block up to MAX_RECORD_SIZE is used to send the data. Users wishing to only send the data in chunks of at most RECORD_SIZE size can do this by defining **STATIC_CHUNKS_ONLY**. This will cause wolfSSL to use I/O buffers which grow up to RECORD_SIZE, which is 128 bytes by default.

Chapter 10: wolfCrypt Usage Reference

wolfCrypt is the cryptography library primarily used by wolfSSL. It is optimized for speed, small footprint, and portability. wolfSSL interchanges with other cryptography libraries as required.

Types used in the examples:

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

10.1 Hash Functions

10.1.1 MD4

NOTE: MD4 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD4 include the MD4 header "wolfssl/wolfcrypt/md4.h". The structure to use is **Md4**, which is a typedef. Before using, the hash initialization must be done with the ***wc_InitMd4()*** call. Use ***wc_Md4Update()*** to update the hash and ***wc_Md4Final()*** to retrieve the final hash.

```
byte md4sum[MD4_DIGEST_SIZE];
byte buffer[1024];
/* fill buffer with data to hash*/

Md4 md4;
wc_InitMd4(&md4);

wc_Md4Update(&md4, buffer, sizeof(buffer));  /*can be called again
                                                  and again*/
wc_Md4Final(&md4, md4sum);
```

md4sum now contains the digest of the hashed data in buffer.

10.1.2 MD5

NOTE: MD5 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD5 include the MD5 header "wolfssl/wolfcrypt/md5.h". The structure to use is **Md5**, which is a typedef. Before using, the hash initialization must be done with the ***wc_InitMd5()*** call. Use ***wc_Md5Update()*** to update the hash and ***wc_Md5Final()*** to retrieve the final hash

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

Md5 md5;
wc_InitMd5(&md5);

wc_Md5Update(&md5, buffer, sizeof(buffer));  /*can be called again
                                                  and again*/
wc_Md5Final(&md5, md5sum);
```

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

10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512

To use SHA include the SHA header "wolfssl/wolfcrypt/sha.h". The structure to use is **Sha**, which is a typedef. Before using, the hash initialization must be done with the ***wc_InitSha()*** call. Use ***wc_ShaUpdate()*** to update the hash and ***wc_ShaFinal()*** to retrieve the final hash:

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

Sha sha;
wc_InitSha(&sha);

wc_ShaUpdate(&sha, buffer, sizeof(buffer));  /*can be called again
                                               and again*/
wc_ShaFinal(&sha, shaSum);
```

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

To use either SHA-224, SHA-256, SHA-384, or SHA-512, follow the same steps as shown above, but use either the "wolfssl/wolfcrypt/sha256.h" or "wolfssl/wolfcrypt/sha512.h" (for both SHA-384 and SHA-512). The SHA-256, SHA-384, and SHA-512 functions are named similarly to the SHA functions.

For **SHA-224**, the functions InitSha224(), Sha224Update(), and Sha224Final() will be used with the structure Sha224.

For **SHA-256**, the functions InitSha256(), Sha256Update(), and Sha256Final() will be used with the structure Sha256.

For **SHA-384**, the functions InitSha384(), Sha384Update(), and Sha384Final() will be used with the structure Sha384.

For **SHA-512**, the functions InitSha512(), Sha512Update(), and Sha512Final() will be used with the structure Sha512.

10.1.4 BLAKE2b

To use BLAKE2b (a SHA-3 finalist) include the BLAKE2b header "wolfssl/wolfcrypt/

blake2.h". The structure to use is **Blake2b**, which is a typedef. Before using, the hash initialization must be done with the ***wc_InitBlake2b()*** call. Use ***wc_Blake2bUpdate()*** to update the hash and ***wc_Blake2bFinal()*** to retrieve the final hash:

```
byte digest[64];
byte input[64];       /*fill input with data to hash*/

Blake2b b2b;
wc_InitBlake2b(&b2b, 64);

wc_Blake2bUpdate(&b2b, input, sizeof(input));
wc_Blake2bFinal(&b2b, digest, 64);
```

The second parameter to wc_InitBlake2b() should be the final digest size. *digest* now contains the digest of the hashed data in buffer.

Example usage can be found in the wolfCrypt test application (wolfcrypt/test/test.c), inside the blake2b_test() function.

10.1.5 RIPEMD-160

To use RIPEMD-160, include the header "wolfssl/wolfcrypt/ripemd.h". The structure to use is **RipeMd**, which is a typedef. Before using, the hash initialization must be done with the ***wc_InitRipeMd()*** call. Use ***wc_RipeMdUpdate()*** to update the hash and ***wc_RipeMdFinal()*** to retrieve the final hash

```
byte ripeMdSum[RIPEMD_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

RipeMd ripemd;
wc_InitRipeMd(&ripemd);

wc_RipeMdUpdate(&ripemd, buffer, sizeof(buffer));  /*can be called
                                                     again and again*/
wc_RipeMdFinal(&ripemd, ripeMdSum);
```

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

10.2 Keyed Hash Functions

10.2.1 HMAC

wolfCrypt currently provides HMAC for message digest needs. The structure **Hmac** is found in the header "wolfssl/wolfcrypt/hmac.h". HMAC initialization is done with ***wc_HmacSetKey()***. 5 different types are supported with HMAC: MD5, SHA, SHA-256, SHA-384, and SHA-512. Here's an example with SHA-256.

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

wc_HmacSetKey(&hmac, SHA256, key, sizeof(key));
wc_HmacUpdate(&hmac, buffer, sizeof(buffer));
wc_HmacFinal(&hmac, hmacDigest);
```

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

10.2.2 GMAC

wolfCrypt also provides GMAC for message digest needs. The structure **Gmac** is found in the header "wolfssl/wolfcrypt/aes.h", as it is an application AES-GCM. GMAC initialization is done with ***wc_GmacSetKey()***.

```
Gmac gmac;
byte key[16];         /*fill key with keying material*/
byte iv[12];          /*fill iv with an initialization vector*/
byte buffer[2048];    /*fill buffer with data to digest*/
byte gmacDigest[16];

wc_GmacSetKey(&gmac, key, sizeof(key));
wc_GmacUpdate(&gmac, iv, sizeof(iv), buffer, sizeof(buffer),
     gmacDigest, sizeof(gmacDigest));
```

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

10.2.3 Poly1305

wolfCrypt also provides Poly1305 for message digest needs. The structure **Poly1305** is found in the header "wolfssl/wolfcrypt/poly1305.h". Poly1305 initialization is done with ***wc_Poly1305SetKey()***. The process of setting a key in Poly1305 should be done again, with a new key, when next using Poly1305 after wc_Poly1305Final() has been called.

```
Poly1305    pmac;
byte        key[32];        /*fill key with keying material*/
byte        buffer[2048];   /*fill buffer with data to digest*/
byte        pmacDigest[16];

wc_Poly1305SetKey(&pmac, key, sizeof(key));
wc_Poly1305Update(&pmac, buffer, sizeof(buffer));
wc_Poly1305Final(&pmac, pmacDigest);
```

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

10.3 Block Ciphers

10.3.1 AES

wolfCrypt provides support for AES with key sizes of 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits). These key size selections can be set at compile-time through the macros `NO_AES_128`, `NO_AES_192`, and `NO_AES_256`. Supported AES modes include CBC, CTR, GCM, and CCM-8.

CBC mode is supported for both encryption and decryption and is provided through the **wc_AesSetKey**(), **wc_AesCbcEncrypt()** and **wc_AesCbcDecrypt()** functions. Please include the header "wolfssl/wolfcrypt/aes.h" to use AES. AES has a block size of 16 bytes and the IV should also be 16 bytes. Function usage is usually as follows:

```
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*/
wc_AesSetKey(&enc, key, sizeof(key), iv, AES_ENCRYPTION);
wc_AesCbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wolfCrypt also supports CTR (Counter), GCM (Galois/Counter), and CCM-8 (Counter with CBC-MAC) modes of operation for AES. When using these modes, like CBC, include the "wolfssl/wolfcrypt/aes.h" header.

CTR mode is available for encryption through the **wc_AesCtrEncrypt()** function.

GCM mode is available for both encryption and decryption through the **wc_AesGcmSetKey**(), **wc_AesGcmEncrypt**(), and **wc_AesGcmDecrypt**() functions. For a usage example, see the aesgcm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CCM-8 mode is supported for both encryption and decryption through the **wc_AesCcmSetKey**(), **wc_AesCcmEncrypt**(), and **wc_AesCcmDecrypt**() functions. For a usage example, see the aesccm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

10.3.2 DES and 3DES

wolfCrypt provides support for DES and 3DES (Des3 since 3 is an invalid leading C identifier). To use these include the header "wolfssl/wolfcrypt/des.h". The structures you can use are **Des** and **Des3**. Initialization is done through ***wc_Des_SetKey()*** or ***wc_Des3_SetKey()***. CBC encryption/decryption is provided through ***wc_Des_CbcEnrypt()*** / ***wc_Des_CbcDecrypt()*** and ***wc_Des3_CbcEncrypt()*** / ***wc_Des3_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. If 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*/
wc_Des3_SetKey(&enc, key, iv, DES_ENCRYPTION);
wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
wc_Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.3.3 Camellia

wolfCrypt provides support for the Camellia block cipher. To use Camellia include the header "wolfssl/wolfcrypt/camellia.h". The structure you can use is called **Camellia**. Initialization is done through ***wc_CamelliaSetKey()***. CBC encryption/decryption is provided through ***wc_CamelliaCbcEnrypt()*** and ***wc_CamelliaCbcDecrypt()*** while direct encryption/decryption is provided through **wc_CamelliaEncryptDirect()** and **wc_CamelliaDecryptDirect()**.

For usage examples please see the camellia_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

10.4 Stream Ciphers

10.4.1 ARC4

The most common stream cipher used on the Internet is ARC4. wolfCrypt supports it through the header "wolfssl/wolfcrypt/arc4.h". Usage is simpler than block ciphers because there is no block size and the key length can be any length. The following is a typical usage of ARC4.

```
Arc4 enc;
Arc4 dec;

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

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

/*encrypt*/
wc_Arc4SetKey(&enc, key, sizeof(key));
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Arc4SetKey(&dec, key, sizeof(key));
wc_Arc4Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.2 RABBIT

A newer stream cipher gaining popularity is RABBIT. This stream cipher can be used through wolfCrypt by including the header "wolfssl/wolfcrypt/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*/
wc_RabbitSetKey(&enc, key, iv);       /*iv can be a NULL pointer*/
wc_RabbitProcess(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_RabbitSetKey(&dec, key, iv);
wc_RabbitProcess(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.3 HC-128

Another stream cipher in current use is HC-128, which is even faster than RABBIT (about 5 times faster than ARC4). To use it with wolfCrypt, please include the header "wolfssl/wolfcrypt/hc128.h". HC-128 also uses 16-byte keys (128 bits) but uses 16-byte IVs (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*/
wc_Hc128_SetKey(&enc, key, iv);       /*iv can be a NULL pointer*/
wc_Hc128_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Hc128_SetKey(&dec, key, iv);
wc_Hc128_Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.4 ChaCha

ChaCha with 20 rounds is slightly faster than ARC4 while maintaining a high level of security. To use it with wolfCrypt, please include the header "wolfssl/wolfcrypt/chacha.h". ChaCha typically uses 32 byte keys (256 bit) but can also use 16 byte keys (128 bits).

```
CHACHA enc;
CHACHA dec;

const byte key[] = {  /*some key 32 bytes*/};
const byte iv[]  = {  /*some iv 12 bytes*/ };

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

/*encrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter); /*counter is the start block
                                      counter is usually set as 0*/
wc_Chacha_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter);
wc_Chacha_Process(&enc, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

Chacha_SetKey only needs to be set once but for each packet of information sent Chacha_SetIV must be called with a new iv (nonce). Counter is set as an argument to allow for partially decrypting/encrypting information by starting at a different block when performing the encrypt/decrypt process, but in most cases is set to 0. **ChaCha should not be used without a mac algorithm (e.g. Poly1305 , hmac).**

10.5 Public Key Cryptography

10.5.1 RSA

wolfCrypt provides support for RSA through the header "wolfssl/wolfcrypt/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;

wc_InitRng(&rng);

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

Now '*out*' holds the ciphertext from the plain text '*in*'. ***wc_RsaPublicEncrypt()*** will return the length in bytes written to out or a negative number in case of an error. ***wc_RsaPublicEncrypt()*** needs a 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 ***wc_RsaEncryptSize()*** which will return the number of bytes that a successful call to ***wc_RsaPublicEnrypt()*** will write.

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

```
void wc_ErrorString(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*/

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

byte plain[128];
word32 plainSz = wc_RsaPrivateDecrypt(out, outLen, plain,
                              sizeof(plain), &rsaPrivateKey);
```

Now plain will hold plainSz bytes or an error code. For complete examples of each type in wolfCrypt please see the file wolfcrypt/test/test.c. Note that the wc_RsaPrivateKeyDecode function only accepts keys in raw **DER** format.

10.5.2 DH (Diffie-Hellman)

wolfCrypt provides support for Diffie-Hellman through the header "wolfssl/wolfrypt/dh.h". The Diffie-Hellman key exchange algorithm allows two parties to establish a shared secret key. Usage is usually similar to the following example, where **sideA** and **sideB** designate the two parties.

In the following example, **dhPublicKey** contains the Diffie-Hellman public parameters signed by a Certificate Authority (or self-signed). **privA** holds the generated private key for sideA, **pubA** holds the generated public key for sideA, and **agreeA** holds the mutual key that both sides have agreed on.

```
DhKey dhPublicKey;
word32 idx = 0;  /*where to start reading into the
                   publicKeyBuffer*/
word32 pubASz, pubBSz, agreeASz;
byte   tmp[1024];
RNG    rng;

byte privA[128];
byte pubA[128];
byte agreeA[128];

wc_InitDhKey(&dhPublicKey);
```

```
byte publicKeyBuffer[] = { /*holds the raw data from the public key
                             parameters, maybe from a file like
                             dh1024.der*/ }
wc_DhKeyDecode(tmp, &idx, &dhPublicKey, publicKeyBuffer);
wc_InitRng(&rng);  /*Initialize random number generator*/
```

wc_DhGenerateKeyPair() will generate a public and private DH key based on the initial public parameters in dhPublicKey.

```
wc_DhGenerateKeyPair(&dhPublicKey, &rng, privA, &privASz,
                      pubA, &pubASz);
```

After sideB sends their public key (**pubB**) to sideA, sideA can then generate the mutually-agreed key(**agreeA**) using the **wc_DhAgree()** function.

```
wc_DhAgree(&dhPublicKey, agreeA, &agreeASz, privA, privASz,
           pubB, pubBSz);
```

Now, **agreeA** holds sideA's mutually-generated key (of size **agreeASz** bytes). The same process will have been done on sideB.

For a complete example of Diffie-Hellman in wolfCrypt, see the file wolfcrypt/test/test.c.

10.5.3 EDH (Ephemeral Diffie-Hellman)

A wolfSSL server can do Ephemeral Diffie-Hellman. No build changes are needed to add this feature, though an application will have to register the ephemeral group parameters on the server side to enable the EDH cipher suites. A new API can be used to do this:

```
int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p,
                     int pSz,unsigned char* g,int gSz);
```

The example server and echoserver use this function from **SetDH()**.

10.5.4 DSA (Digital Signature Algorithm)

wolfCrypt provides support for DSA and DSS through the header "wolfssl/wolfcrypt/dsa.h". DSA allows for the creation of a digital signature based on a given data hash. DSA uses the SHA hash algorithm to generate a hash of a block of data, then signs that hash using the signer's private key. Standard usage is similar to the following.

We first declare our DSA key structure (**key**), initialize our initial message (**message**) to be signed, and initialize our DSA key buffer (**dsaKeyBuffer**).

```
DsaKey key;
Byte   message[]      = { /*message data to sign*/ }
byte   dsaKeyBuffer[] = { /*holds the raw data from the DSA key,
                            maybe from a file like dsa512.der*/ }
```

We then declare our SHA structure (**sha**), random number generator (**rng**), array to store our SHA hash (**hash**), array to store our signature (**signature**), **idx** (to mark where to start reading in our dsaKeyBuffer), and an int (**answer**) to hold our return value after verification.

```
Sha    sha;
RNG    rng;
byte   hash[SHA_DIGEST_SIZE];
byte   signature[40];
word32 idx = 0;
int    answer;
```

Set up and create the SHA hash. For more information on wolfCrypt's SHA algorithm, see section 10.1.3. The SHA hash of "**message**" is stored in the variable "**hash**".

```
wc_InitSha(&sha);
wc_ShaUpdate(&sha, message, sizeof(message));
wc_ShaFinal(&sha, hash);
```

Initialize the DSA key structure, populate the structure key value, and initialize the random number generator (**rng**).

```
wc_InitDsaKey(&key);
wc_DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key,
                       sizeof(dsaKeyBuffer));
wc_InitRng(&rng);
```

The **wc_DsaSign()** function creates a signature (**signature**) using the DSA private key, hash value, and random number generator.

```
wc_DsaSign(hash, signature, &key, &rng);
```

To verify the signature, use **wc_DsaVerify()**. If verification is successful, answer will be equal to “**1**”. Once finished, free the DSA key structure using **wc_FreeDsaKey()**.

```
wc_DsaVerify(hash, signature, &key, &answer);
wc_FreeDsaKey(&key);
```

Chapter 11: SSL Tutorial

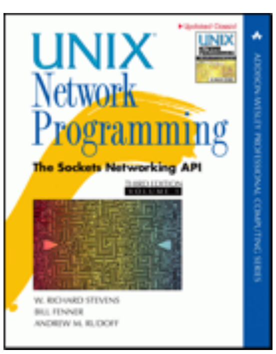
11.1 Introduction

The wolfSSL (formerly CyaSSL) embedded SSL library can easily be integrated into your existing application or device to provide enhanced communication security through the addition of SSL and TLS. wolfSSL has been targeted at embedded and RTOS environments, and as such, offers a minimal footprint while maintaining excellent performance. Minimum build sizes for wolfSSL range between 20-100kB depending on the selected build options and platform being used.

The goal of this tutorial is to walk through the integration of SSL and TLS into a simple application. Hopefully the process of going through this tutorial will also lead to a better understanding of SSL in general. This tutorial uses wolfSSL in conjunction with simple echoserver and echoclient examples to keep things as simple as possible while still demonstrating the general procedure of adding SSL support to an application. The echoserver and echoclient examples have been taken from the popular book titled “**Unix Network Programming, Volume 1, 3rd Edition**” by Richard Stevens, Bill Fenner, and Andrew Rudoff.

This tutorial assumes that the reader is comfortable with editing and compiling C code using the GNU GCC compiler as well as familiar with the concepts of public key encryption. Please note that access to the Unix Network Programming book is not required for this tutorial.

Examples Used in this Tutorial
echoclient - Figure 5.4, Page 124
echoserver - Figure 5.12, Page 139



Unix Network Programming
Volume 1, 3rd Edition

www.unpbook.com

11.2 Quick Summary of SSL/TLS

TLS (Transport Layer Security) and **SSL** (Secure Sockets Layer) are cryptographic protocols that allow for secure communication across a number of different transport protocols. The primary transport protocol used is TCP/IP. The most recent version of SSL/TLS is TLS 1.3. wolfSSL supports SSL 3.0, TLS 1.0, 1.1, 1.2, 1.3 in addition to DTLS 1.0 and 1.2.

SSL and TLS sit between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the underlying transport medium. Application protocols are layered on top of SSL and can include protocols such as HTTP, FTP, and SMTP. A diagram of how SSL fits into the OSI model, as well as a simple diagram of the SSL handshake process can be found in Appendix A.

11.3 Getting the Source Code

All of the source code used in this tutorial can be downloaded from the wolfSSL website, specifically from the following location. The download contains both the original and completed source code for both the echoserver and echoclient used in this tutorial. Specific contents are listed below the link.

http://www.wolfssl.com/documentation/ssl-tutorial-2.2.zip

```
The downloaded ZIP file has the following structure:

/finished_src
      /echoclient      (Completed echoclient code)
      /echoserver      (Completed echoserver code)
      /include         (Modified unp.h)
      /lib             (Library functions)
/original_src
      /echoclient      (Starting echoclient code)
      /echoserver      (Starting echoserver code)
      /include         (Modified unp.h)
      /lib             (Library functions)
README
```

11.4 Base Example Modifications

This tutorial, and the source code that accompanies it, have been designed to be as portable as possible across platforms. Because of this, and because we want to focus on how to add SSL and TLS into an application, the base examples have been kept as simple as possible. Several modifications have been made to the examples taken from Unix Network Programming in order to either remove unnecessary complexity or increase the range of platforms supported. If you believe there is something we could do to increase the portability of this tutorial, please let us know at support@wolfssl.com.

The following is a list of modifications that were made to the original echoserver and echoclient examples found in the above listed book.

Modifications to the echoserver (tcpserv04.c)

- Removed call to the Fork() function because fork() is not supported by Windows. The result of this is an echoserver which only accepts one client simultaneously. Along with this removal, Signal handling was removed.

- Moved str_echo() function from str_echo.c file into tcpserv04.c file

- Added a printf statement to view the client address and the port we have connected through:

```
printf("Connection from %s, port %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
        ntohs(cliaddr.sin_port));
```

- Added a call to setsockopt() after creating the listening socket to eliminate the “Address already in use” bind error.

- Minor adjustments to clean up newer compiler warnings

Modifications to the echoclient (tcpcli01.c)

- Moved str_cli() function from str_cli.c file into tcpcli01.c file

- Minor adjustments to clean up newer compiler warnings

Modifications to unp.h header

- This header was simplified to contain only what is needed for this example.

Please note that in these source code examples, certain functions will be capitalized. For example, Fputs() and Writen(). The authors of Unix Network Programming have written custom wrapper functions for normal functions in order to cleanly handle error checking. For a more thorough explanation of this, please see **Section 1.4** (page 11) in the *Unix Network Programming* book.

11.5 Building and Installing wolfSSL

Before we begin, download the example code (echoserver and echoclient) from the Getting the Source Code section, above. This section will explain how to download, configure, and install the wolfSSL embedded SSL library on your system.

You will need to download and install the most recent version of wolfSSL from the wolfSSL download page.

For a full list of available build options, see the Building wolfSSL guide. wolfSSL was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please feel free to ask for support on the wolfSSL product support forums.

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other *nix like systems, you can use the autoconf system. For windows-specific instructions, please refer to the Building wolfSSL section of the wolfSSL Manual. To configure and build wolfSSL, run the following two commands from the terminal. Any desired build options may be appended to ./configure (ex: ./configure –enable-opensslextra):

```
./configure
make
```

To install wolfSSL, run:

```
sudo make install
```

This will install wolfSSL headers into /usr/local/include/wolfssl and the wolfSSL libraries into /usr/local/lib on your system. To test the build, run the testsuite application from the wolfSSL root directory:

```
./testsuite/testsuite.test
```

A set of tests will be run on wolfCrypt and wolfSSL to verify it has been installed correctly. After a successful run of the testsuite application, you should see output similar to the following:

```
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES-GCM  test passed!
RANDOM   test passed!
RSA      test passed!
DH       test passed!
ECC      test passed!
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:ECDHE-
RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-
SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-
AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-
SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-
SHA384:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:DHE-RSA-
CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-OLD:ECDHE-ECDSA-CHACHA20-
POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /tmp/
output-N0Xq9c
```

```
All tests passed!
```

Now that wolfSSL has been installed, we can begin modifying the example code to add SSL functionality. We will first begin by adding SSL to the echoclient and subsequently move on to the echoserver.

11.6 Initial Compilation

To compile and run the example echoclient and echoserver code from the SSL Tutorial source bundle, you can use the included Makefiles. Change directory (cd) to either the echoclient or echoserver directory and run:

```
make
```

This will compile the example code and produce an executable named either echoserver or echoclient depending on which one is being built. The GCC command which is used in the Makefile can be seen below. If you want to build one of the examples without using the supplied Makefile, change directory to the example directory and replace tcpcli01.c (echoclient) or tcpserv04.c (echoserver) in the following command with correct source file for the example:

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

This will compile the current example into an executable, creating either an “echoserver” or “echoclient” application. To run one of the examples after it has been compiled, change your current directory to the desired example directory and start the application. For example, to start the echoserver use:

```
./echoserver
```

You may open a second terminal window to test the echoclient on your local host and you will need to supply the IP address of the server when starting the application, which in our case will be 127.0.0.1. Change your current directory to the “echoclient” directory and run the following command. Note that the echoserver must already be running:

```
./echoclient 127.0.0.1
```

Once you have both the echoserver and echoclient running, the echoserver should echo back any input that it receives from the echoclient. To exit either the echoserver or echoclient, use [Ctrl + C] to quit the application. Currently, the data being echoed back and forth between these two examples is being sent in the clear - easily allowing anyone with a little bit of skill to inject themselves in between the client and server and listen to your communication.

11.7 Libraries

The wolfSSL library, once compiled, is named libwolfssl, and unless otherwise configured the wolfSSL build and install process creates only a shared library under the following directory. Both shared and static libraries may be enabled or disabled by using the appropriate build options:

```
/usr/local/lib
```

The first step we need to do is link the wolfSSL library to our example applications. Modifying the GCC command (using the echoserver as an example), gives us the following new command. Since wolfSSL installs header files and libraries in standard locations, the compiler should be able to find them without explicit instructions (using -l or -L). Note that by using -lwolfssl the compiler will automatically choose the correct type of library (static or shared):

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lwolfssl
```

11.8 Headers

The first thing we will need to do is include the wolfSSL native API header in both the client and the server. In the tcpcli01.c file for the client and the tcpserv04.c file for the server add the following line near the top:

```
#include <wolfssl/ssl.h>
```

11.9 Startup/Shutdown

Before we can use wolfSSL in our code, we need to initialize the library and the WOLFSSL_CTX. wolfSSL is initialized by calling wolfSSL_Init(). This must be done first before anything else can be done with the library.

The WOLFSSL_CTX structure (wolfSSL Context) contains global values for each SSL connection, including certificate information. A single WOLFSSL_CTX can be used with any number of WOLFSSL objects created. This allows us to load certain information, such as a list of trusted CA certificates only once.

To create a new WOLFSSL_CTX, use wolfSSL_CTX_new(). This function requires an argument which defines the SSL or TLS protocol for the client or server to use. There are several options for selecting the desired protocol. wolfSSL currently supports SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, DTLS 1.0, and DTLS 1.2. Each of these protocols have a corresponding function that can be used as an argument to wolfSSL_CTX_new(). The possible client and server protocol options are shown below. SSL 2.0 is not supported by wolfSSL because it has been insecure for several years.

EchoClient:

```
wolfSSLv3_client_method();     // SSL 3.0
wolfTLSv1_client_method();     // TLS 1.0
wolfTLSv1_1_client_method();   // TLS 1.1
wolfTLSv1_2_client_method();   // TLS 1.2
wolfSSLv23_client_method();    // Use highest version possible from
                               // SSLv3 - TLS 1.2
wolfDTLSv1_client_method();     // DTLS 1.0
wolfDTLSv1_2_client_method();   // DTLS 1.2
```

EchoServer:

```
wolfSSLv3_server_method();     // SSLv3
wolfTLSv1_server_method();     // TLSv1
wolfTLSv1_1_server_method();   // TLSv1.1
wolfTLSv1_2_server_method();   // TLSv1.2
wolfSSLv23_server_method();    // Allow clients to connect with
                               // TLSv1+
wolfDTLSv1_server_method();    // DTLS
wolfDTLSv1_2_server_method();   // DTLS 1.2
```

We need to load our CA (Certificate Authority) certificate into the WOLFSSL_CTX so that the when the echoclient connects to the echoserver, it is able to verify the server's identity. To load the CA certificates into the WOLFSSL_CTX, use wolfSSL_CTX_load_verify_locations(). This function requires three arguments: a WOLFSSL_CTX pointer, a certificate file, and a path value. The path value points to a directory which should contain CA certificates in PEM format. When looking up certificates, wolfSSL will look at the certificate file value before looking in the path location. In this case, we don't need to specify a certificate path because we will specify one CA file - as such we use the value 0 for the path argument. The wolfSSL_CTX_load_verify_locations function returns either SSL_SUCCESS or SSL_FAILURE:

```
wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file, const
char* path)
```

Putting these things together (library initialization, protocol selection, and CA certificate), we have the following. Here, we choose to use TLS 1.2:

EchoClient:

```
WOLFSSL_CTX* ctx;

wolfSSL_Init();/* Initialize wolfSSL */

/* Create the WOLFSSL_CTX */
if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) == NULL){
     fprintf(stderr, "wolfSSL_CTX_new error.\n");
     exit(EXIT_FAILURE);
}

/* Load CA certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx,"../certs/ca-cert.pem",0) !=
    SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/ca-cert.pem, please check
            the file.\n");
    exit(EXIT_FAILURE);
}
```

EchoServer:

When loading certificates into the WOLFSSL_CTX, the server certificate and key file should be loaded in addition to the CA certificate. This will allow the server to send the client its certificate for identification verification:

```
WOLFSSL_CTX* ctx;

wolfSSL_Init();/* Initialize wolfSSL */

/* Create the WOLFSSL_CTX */
if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) == NULL){
     fprintf(stderr, "wolfSSL_CTX_new error.\n");
     exit(EXIT_FAILURE);
}

/* Load CA certificates into CYASSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx, "../certs/ca-cert.pem", 0) !=
         SSL_SUCCESS) {
     fprintf(stderr, "Error loading ../certs/ca-cert.pem, "
         "please check the file.\n");
     exit(EXIT_FAILURE);
}


/* Load server certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_use_certificate_file(ctx,"../certs/server-cert.pem",
          SSL_FILETYPE_PEM) != SSL_SUCCESS){
    fprintf(stderr, "Error loading ../certs/server-cert.pem, please
          check the file.\n");
    exit(EXIT_FAILURE);
}

/* Load keys */
if (wolfSSL_CTX_use_PrivateKey_file(ctx,"../certs/server-key.pem",
          SSL_FILETYPE_PEM) != SSL_SUCCESS){
    fprintf(stderr, "Error loading ../certs/server-key.pem, please check
          the file.\n");
    exit(EXIT_FAILURE);
}
```

The code shown above should be added to the beginning of tcpcli01.c and tcpserv04.c, after both the variable definitions and the check that the user has started the client with an IP address (client). A version of the finished code is included in the SSL tutorial ZIP file for reference.

Now that wolfSSL and the WOLFSSL_CTX have been initialized, make sure that the

WOLFSSL_CTX object and the wolfSSL library are freed when the application is completely done using SSL/TLS. In both the client and the server, the following two lines should be placed at the end of the main() function (in the client right before the call to exit()):

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

11.10 WOLFSSL Object

EchoClient

A WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session. In the echoclient example, we will do this after the call to Connect(), shown below:

```
/* Connect to socket file descriptor */
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

Directly after connecting, create a new WOLFSSL object using the wolfSSL_new() function. This function returns a pointer to the WOLFSSL object if successful or NULL in the case of failure. We can then associate the socket file descriptor (sockfd) with the new WOLFSSL object (ssl):

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, sockfd);
```

One thing to notice here is we haven't made a call to the wolfSSL_connect() function. wolfSSL_connect() initiates the SSL/TLS handshake with the server, and is called during wolfSSL_read() if it hasn't been called previously. In our case, we don't explicitly call wolfSSL_connect(), as we let our first wolfSSL_read() do it for us.

EchoServer

At the end of the for loop in the main method, insert the WOLFSSL object and associate the socket file descriptor (connfd) with the WOLFSSL object (ssl), just as with the client:

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if ( (ssl = wolfSSL_new(ctx)) == NULL) {
                    fprintf(stderr, "wolfSSL_new error.\n");

                    exit(EXIT_FAILURE);
             }

wolfSSL_set_fd(ssl, connfd);
```

A WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session.

Create a new WOLFSSL object using the **wolfSSL_new()** function. This function returns a pointer to the WOLFSSL object if successful or NULL in the case of failure. We can then associate the socket file descriptor (**sockfd**) with the new WOLFSSL object (**ssl**):

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, sockfd);
```

11.11 Sending/Receiving Data

EchoClient

The next step is to begin sending data securely. Take note that in the echoclient example, the main() function hands off the sending and receiving work to str_cli(). The str_cli() function is where our function replacements will be made. First we need access to our WOLFSSL object in the str_cli() function, so we add another argument and pass the ssl variable to str_cli(). Because the WOLFSSL object is now going to be used inside of the str_cli() function, we remove the sockfd parameter. The new str_cli() function signature after this modification is shown below:

```
void str_cli(FILE *fp, WOLFSSL* ssl)
```

In the main() function, the new argument (ssl) is passed to str_cli():

```
str_cli(stdin, ssl);
```

Inside the str_cli() function, Writen() and Readline() are replaced with calls to wolfSSL_write() and wolfSSL_read() functions, and the WOLFSSL object (ssl) is used instead of the original file descriptor(sockfd). The new str_cli() function is shown below. Notice that we now need to check if our calls to wolfSSL_write and wolfSSL_read were successful.

The authors of the Unix Programming book wrote error checking into their Writen() function which we must make up for after it has been replaced. We add a new int variable, "n", to monitor the return value of wolfSSL_read and before printing out the contents of the buffer, recvline, the end of our read data is marked with a '\0':

```
void
str_cli(FILE *fp, WOLFSSL* ssl)
{
    char  sendline[MAXLINE], recvline[MAXLINE];
    int   n = 0;

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        if(wolfSSL_write(ssl, sendline, strlen(sendline)) !=
                           strlen(sendline)){
            err_sys("wolfSSL_write failed");
        }

        if ((n = wolfSSL_read(ssl, recvline, MAXLINE)) <= 0)
            err_quit("wolfSSL_read error");

        recvline[n] = '\0';
        Fputs(recvline, stdout);
    }
}
```

The last thing to do is free the WOLFSSL object when we are completely done with it. In the main() function, right before the line to free the WOLFSSL_CTX, call to wolfSSL_free():

```
str_cli(stdin, ssl);

wolfSSL_free(ssl);       /* Free WOLFSSL object */
wolfSSL_CTX_free(ctx);   /* Free WOLFSSL_CTX object */
wolfSSL_Cleanup();       /* Free wolfSSL */
```

EchoServer

The echo server makes a call to str_echo() to handle reading and writing (whereas the client made a call to str_cli()). As with the client, modify str_echo() by replacing the sockfd parameter with a WOLFSSL object (ssl) parameter in the function signature:

```
void str_echo(WOLFSSL* ssl)
```

Replace the calls to Read() and Writen() with calls to the wolfSSL_read() and wolfSSL_write() functions. The modified str_echo() function, including error checking of return values, is shown below. Note that the type of the variable “n” has been changed from ssize_t to int in order to accommodate for the change from read() to wolfSSL_read():

```
void
str_echo(WOLFSSL* ssl)
{
    int n;
    char buf[MAXLINE];

    while ( (n = wolfSSL_read(ssl, buf, MAXLINE)) > 0) {
        if(wolfSSL_write(ssl, buf, n) != n) {
            err_sys("wolfSSL_write failed");
        }
    }

    if( n < 0 )
        printf("wolfSSL_read error = %d\n", wolfSSL_get_error(ssl,n));

    else if( n == 0 )
        printf("The peer has closed the connection.\n");
}
```

In main() call the str_echo() function at the end of the for loop (soon to be changed to a while loop). After this function, inside the loop, make calls to free the WOLFSSL object and close the connfd socket:

```
str_echo(ssl);                          /* process the request */

wolfSSL_free(ssl);                      /* Free WOLFSSL object */
Close(connfd);
```

We will free the ctx and cleanup before the call to exit.

11.12 Signal Handling

Echoclient / Echoserver

In the echoclient and echoserver, we will need to add a signal handler for when the user closes the app by using “Ctrl+C”. The echo server is continually running in a loop. Because of this, we need to provide a way to break that loop when the user presses “Ctrl+C”. To do this, the first thing we need to do is change our loop to a while loop which terminates when an exit variable (cleanup) is set to true.

First, define a new static int variable called cleanup at the top of tcpserv04.c right after the #include statements:

```
static int cleanup;  /* To handle shutdown */
```

Modify the echoserver loop by changing it from a for loop to a while loop:

```
while(cleanup != 1)
{
    /* echo server code here */
}
```

For the echoserver we need to disable the operating system from restarting calls which were being executed before the signal was handled after our handler has finished. By disabling these, the operating system will not restart calls to accept() after the signal has been handled. If we didn’t do this, we would have to wait for another client to connect and disconnect before the echoserver would clean up resources and exit. To define the signal handler and turn off SA_RESTART, first define act and oact structures in the echoserver's main() function:

```
struct sigaction     act, oact;
```

Insert the following code after variable declarations, before the call to wolfSSL_Init() in the main function:

```
    /* Signal handling code */
    struct sigaction act, oact;         /* Declare the sigaction structs
*/
    act.sa_handler = sig_handler;       /* Tell act to use sig_handler
*/
    sigemptyset(&act.sa_mask);          /* Tells act to exclude all sa_mask
*
                                         * signals during execution of
*
                                         * sig_handler.
*/
    act.sa_flags = 0;                   /* States that act has a special
*
                                         * flag of 0
*/
    sigaction(SIGINT, &act, &oact);     /* Tells the program to use (o)act
*
                                         * on a signal or interrupt
*/
```

The echoserver's sig_handler function is shown below:

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    cleanup = 1;
    return;
}
```

That's it - the echoclient and echoserver are now enabled with TLSv1.2!!
What we did:

- Included the wolfSSL headers
- Initialized wolfSSL
- Created a WOLFSSL_CTX structure in which we chose what protocol we wanted

to use
- Created a WOLFSSL object to use for sending and receiving data
- Replaced calls to Writen() and Readline() with wolfSSL_write() and wolfSSL_read()
- Freed WOLFSSL, WOLFSSL_CTX
- Made sure we handled client and server shutdown with signal handler

There are many more aspects and methods to configure and control the behavior of your SSL connections. For more detailed information, please see additional wolfSSL documentation and resources.

Once again, the completed source code can be found in the downloaded ZIP file at the top of this section.

11.13 Certificates

For testing purposes, you may use the certificates provided by wolfSSL. These can be found in the wolfSSL download, and specifically for this tutorial, they can be found in the **finished_src** folder.

For production applications, you should obtain correct and legitimate certificates from a trusted certificate authority.

11.14 Conclusion

This tutorial walked through the process of integrating the wolfSSL embedded SSL library into a simple client and server application. Although this example is simple, the same principles may be adpplied for adding SSL or TLS into your own application. The wolfSSL embedded SSL library provides all the features you would need in a compact and efficient package that has been optimized for both size and speed.

Being dual licensed under GPLv2 and standard commercial licensing, you are free to download the wolfSSL source code directly from our website. Feel free to post to our support forums (www.wolfssl.com/forums) with any questions or comments you might have. If you would like more information about our products, please contact info@wolfssl.com.

We welcome any feedback you have on this SSL tutorial. If you believe it could be improved or enhanced in order to make it either more useful, easier to understand, or more portable, please let us know at support@wolfssl.com.

Chapter 12: Best Practices for Embedded Devices

12.1 Creating Private Keys

Embedding a private key into firmware allows anyone to extract the key and turns an otherwise secure connection into something nothing more secure than TCP.

We have a few ideas about creating private keys for SSL enabled devices.

1. Each device acting as a server should have a unique private key, just like in the non-embedded world.

2. If the key can't be placed onto the device before delivery, have it generated during setup.

3. If the device lacks the power to generate it's own key during setup, have the client setting up the device generate the key and send it to the device.

4. If the client lacks the ability to generate a private key, have the client retrieve a unique private key over an SSL/TLS connection from the devices known website (for example).

wolfSSL (formerly CyaSSL) can be used in all of these steps to help ensure an embedded device has a secure unique private key. Taking these steps will go a long ways towards securing the SSL connection itself.

12.2 Digitally Signing and Authenticating with wolfSSL

wolfSSL is a popular tool for digitally signing applications, libraries, or files prior to loading them on embedded devices. Most desktop and server operating systems allow creation of this type of functionality through system libraries, but stripped down embedded operating systems do not. The reason that embedded RTOS environments do not include digital signature functionality is because it has historically not been a requirement for most embedded applications. In today’s world of connected devices and heightened security concerns, digitally signing what is loaded onto your embedded or mobile device has become a top priority.

Examples of embedded connected devices where this requirement was not found in years past include set top boxes, DVR's, POS systems, both VoIP and mobile phones, connected home, and even automobile-based computing systems. Because wolfSSL supports the key embedded and real time operating systems, encryption standards, and authentication functionality, it is a natural choice for embedded systems developers to use when adding digital signature functionality.

Generally, the process for setting up code and file signing on an embedded device are as follows:

1. The embedded systems developer will generate an RSA key pair.
2. A server-side script-based tool is developed
    a. The server side tool will create a hash of the code to be loaded on the device (with SHA-256 for example).
    b. The hash is then digitally signed, also called RSA private encrypt.
    c. A package is created that contains the code along with the digital signature.
3. The package is loaded on the device along with a way to get the RSA public key. The hash is re-created on the device then digitally verified (also called RSA public decrypt) against the existing digital signature.

Benefits to enabling digital signatures on your device include:

1. Easily enable a secure method for allowing third parties to load files to your device.
2. Ensure against malicious files finding their way onto your device.
3. Digitally secure firmware updates
4. Ensure against firmware updates from unauthorized parties

General information on code signing:
http://en.wikipedia.org/wiki/Code_signing

Chapter 13: OpenSSL Compatibility

13.1 Compatibility with OpenSSL

wolfSSL (formerly CyaSSL) provides an OpenSSL compatibility header, **wolfssl/openssl/ssl.h**, in addition to the wolfSSL native API, to ease the transition into using wolfSSL or to aid in porting an existing OpenSSL application over to wolfSSL. For an overview of the OpenSSL Compatibility Layer, please continue reading below. To view the complete set of OpenSSL functions supported by wolfSSL, please see the **wolfssl/openssl/ssl.h** file.

The OpenSSL Compatibility Layer maps a subset of the most commonly-used OpenSSL commands to wolfSSL's native API functions. This should allow for an easy replacement of OpenSSL by wolfSSL in your application or project without changing much code.

Our test beds for OpenSSL compatibility are stunnel and Lighttpd, which means that we build both of them with wolfSSL as a way to test our OpenSSL compatibility API.

Building wolfSSL With Compatibility Layer:

1. Enable with (--enable-opensslextra) or by defining the macro OPENSSL_EXTRA.
    a. ./configure --enable-opensslextra
2. Include <wolfssl/options.h> as first wolfSSL header
3. Header files for migration are located under:
    a. ./wolfssl/openssl/*.h
    b. Ex: <wolfssl/openssl/ssl.h>

13.2 Differences Between wolfSSL and OpenSSL

Many people are curious how wolfSSL compares to OpenSSL and what benefits there are to using an SSL/TLS library that has been optimized to run on embedded platforms. Obviously, OpenSSL is free and presents no initial costs to begin using, but we believe that wolfSSL will provide you with more flexibility, an easier integration of SSL/TLS into your existing platform, current standards support, and much more – all provided under a very easy-to-use license model.

The points below outline several of the main differences between wolfSSL and OpenSSL.

1. With a 20-100 kB build size, wolfSSL is up to 20 times smaller than OpenSSL. wolfSSL is a better choice for resource constrained environments – where every byte matters.

2. wolfSSL is up to date with the most current standards of TLS 1.3 with DTLS. The wolfSSL team is dedicated to continually keeping wolfSSL up-to-date with current standards.

3. wolfSSL offers the best current ciphers and standards available today, including ciphers for streaming media support. In addition, the recently-introduced NTRU cipher allows speed increases of 20-200x over standard RSA.

4. wolfSSL is dual licensed under both the GPLv2 as well as a commercial license, where OpenSSL is available only under their unique license from multiple sources.

5. wolfSSL is backed by an outstanding company who cares about its users and about their security, and is always willing to help. The team actively works to improve and expand wolfSSL. The wolfSSL team is based primarily out of Bozeman, MT, Portland, OR, and Seattle, WA, along with other team members located around the globe.

6. wolfSSL is the leading SSL/TLS library for real time, mobile, and embedded systems by virtue of its breadth of platform support and successful implementations on embedded environments. Chances are we've already been ported to your environment. If not, let us know and we'll be glad to help.

7. wolfSSL offers several abstraction layers to make integrating SSL into your environment and platform as easy as possible. With an OS layer, a custom I/O layer, and a C Standard Library abstraction layer, integration has never been so easy.

8. wolfSSL offers several support packages for wolfSSL. Available directly through phone, email or the wolfSSL product support forums, your questions are answered quickly and accurately to help you make progress on your project as quickly as possible.

13.3 Supported OpenSSL Structures

SSL_METHOD holds SSL version information and is either a client or server method. (Same as WOLFSSL_METHOD in the native wolfSSL API).

SSL_CTX holds context information including certificates. (Same as WOLFSSL_CTX in the native wolfSSL API).

SSL holds session information for a secure connection. (Same as WOLFSSL in the native wolfSSL API).

13.4 Supported OpenSSL Functions

The three structures shown above are usually initialized in the following way:

```
SSL_METHOD* method = SSLv3_client_method();
SSL_CTX* ctx = SSL_CTX_new(method);
SSL* ssl = SSL_new(ctx);
```

This establishes a client side SSL version 3 method, creates a context based on the method, and initializes the SSL session with the context. A server side program is no different except that the **SSL_METHOD** is created using ***SSLv3_server_method()***, or one of the available functions. For a list of supported functions, please see **Section 4.2**. When using the OpenSSL Compatibility layer, the functions in 4.2 should be modified by removing the “wolf” prefix. For example, the native wolfSSL API function:

```
wolfTLSv1_client_method()
```

Becomes

```
TLSv1_client_method()
```

When an SSL connection is no longer needed the following calls free the structures created during initialization.

```
SSL_CTX_free(ctx);
SSL_free(ssl);
```

SSL_CTX_free() has the additional responsibility of freeing the associated

SSL_METHOD. Failing to use the XXX_free() functions will result in a resource leak. Using the system's **free()** instead of the SSL ones results in undefined behavior.

Once an application has a valid SSL pointer from **SSL_new()**, the SSL handshake process can begin. From the client's view, **SSL_connect()** will attempt to establish a secure connection.

```
SSL_set_fd(ssl, sockfd);
SSL_connect(ssl);
```

Before the **SSL_connect()** can be issued, the user must supply wolfSSL with a valid socket file descriptor, sockfd in the example above. sockfd is typically the result of the TCP function **socket()** which is later established using TCP **connect()**. The following creates a valid client side socket descriptor for use with a local wolfSSL server on port 11111, error handling is omitted for simplicity.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(11111);
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
connect(sockfd, (const sockaddr*)&servaddr, sizeof(servaddr));
```

Once a connection is established, the client may read and write to the server. Instead of using the TCP functions **send()** and **receive()**, wolfSSL and yaSSL use the SSL functions **SSL_write()** and **SSL_read()**. Here is a simple example from the client demo:

```
char msg[] = "hello wolfssl!";
int wrote = SSL_write(ssl, msg, sizeof(msg));
char reply[1024];
int read = SSL_read(ssl, reply, sizeof(reply));
reply[read] = 0;
printf("Server response: %s\n", reply);
```

The server connects in the same way except that it uses **SSL_accept()** instead of **SSL_connect()**, analogous to the TCP API. See the server example for a complete server demo program.

13.5 x509 Certificates

Both the server and client can provide wolfSSL with certificates in either **PEM** or **DER**.

Typical usage is like this:

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem",
      SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der",
      SSL_FILETYPE_ASN1);
```

A key file can also be presented to the Context in either format. **SSL_FILETYPE_PEM** signifies the file is PEM formatted while **SSL_FILETYPE_ASN1** declares the file to be in DER format. To verify that the key file is appropriate for use with the certificate the following function can be used:

```
SSL_CTX_check_private_key(ctx);
```

Chapter 14: Licensing

14.1 Open Source

wolfSSL (formerly CyaSSL), yaSSL, wolfCrypt, yaSSH and TaoCrypt software are free software downloads and may be modified to the needs of the user as long as the user adheres to version two of the GPL License. The GPLv2 license can be found on the gnu.org website (http://www.gnu.org/licenses/old-licenses/gpl-2.0.html).

wolfSSH software is a free software download and may be modified to the needs of the user as long as the user adheres to version three of the GPL license. The GPLv3 license can be found on the gnu.org website (https://www.gnu.org/licenses/gpl.html).

14.2 Commercial Licensing

Businesses and enterprises who wish to incorporate wolfSSL products into proprietary appliances or other commercial software products for re-distribution must license commercial versions. Commercial licenses for wolfSSL, yaSSL, and wolfCrypt are available for $5,000 USD per end product or SKU. Licenses are generally issued for one product and include unlimited royalty-free distribution. Custom licensing terms are also available.

Commercial licenses are also available for wolfMQTT and wolfSSH. Please contact licensing@wolfssl.com with inquiries.

14.3 Support Packages

Support packages for wolfSSL products are available on an annual basis directly from wolfSSL. With three different package options, you can compare them side-by-side and choose the package that best fits your specific needs. Please see our Support Packages page (https://www.wolfssl.com/wolfSSL/Support/support_tiers.php) for more details.

Chapter 15: Support and Consulting

15.1 How to Get Support

For general product support, wolfSSL (formerly CyaSSL) maintains an online forum for the wolfSSL product family. Please post to the forums or contact wolfSSL directly with any questions.

wolfSSL (yaSSL) Forums: https://www.wolfssl.com/forums
Email Support: support@wolfssl.com

For information regarding wolfSSL products, questions regarding licensing, or general comments, please contact wolfSSL by emailing **info@wolfssl.com**. For support packages, please see **Chapter 14**.

15.1.1 Bugs Reports and Support Issues

If you are submitting a bug report or asking about a problem, please include the following information with your submission:

1. wolfSSL version number
2. Operating System version
3. Compiler version
4. The exact error you are seeing
5. A description of how we can reproduce or try to replicate this problem

With the above information, we will do our best to resolve your problems. Without this information, it is very hard to pinpoint the source of the problem. wolfSSL values your feedback and makes it a top priority to get back to you as soon as possible.

15.2 Consulting

wolfSSL offers both on and off site consulting - providing feature additions, porting, a Competitive Upgrade Program (see section 15.2.2), and design consulting.

15.2.1 Feature Additions and Porting

We can add additional features that you may need which are not currently offered in our products on a contract or co-development basis. We also offer porting services on our products to new host languages or new operating environments.

15.2.2 Competitive Upgrade Program

We will help you move from an outdated or expensive SSL/TLS library to wolfSSL with low cost and minimal disturbance to your code base.

Program Outline:

1. You need to currently be using a commercial competitor to wolfSSL.
2. You will receive up to one week of on-site consulting to switch out your old SSL library for wolfSSL. Travel expenses are not included.
3. Normally, up to one week is the right amount of time for us to make the replacement in your code and do initial testing. Additional consulting on a replacement is available as needed.
4. You will receive the standard wolfSSL royalty free license to ship with your product.
5. The price is $10,000.

The purpose of this program is to enable users who are currently spending too much on their embedded SSL implementation to move to wolfSSL with ease. If you are interested in learning more, then please contact us at info@wolfssl.com.

15.2.3 Design Consulting

If your application or framework needs to be secured with SSL/TLS but you are uncertain about how the optimal design of a secured system would be structured, we can help!

We offer design consulting for building SSL/TLS security into devices using wolfSSL. Our consultants can provide you with the following services:

1. *Assessment:* An evaluation of your current SSL/TLS implementation. We can give you advice on your current setup and how we think you could improve upon

this by using wolfSSL.

2. *Design:* Looking at your system requirements and parameters, we'll work closely with you to make recommendations on how to implement wolfSSL into your application such that it provides you with optimal security.

If you would like to learn more about design consulting for building SSL into your application or device, please contact info@wolfssl.com for more information.

Chapter 16: wolfSSL (formerly CyaSSL) Updates

16.1 Product Release Information

We regularly post update information on Twitter. For additional release information, you can keep track of our projects on GitHub, follow us on Facebook, or follow our daily blog.

wolfSSL on GitHub	https://www.github.com/wolfssl/wolfssl
wolfSSL on Twitter	http://twitter.com/wolfSSL
wolfSSL on Facebook	http://www.facebook.com/wolfSSL
wolfSSL on Reddit	https://www.reddit.com/r/wolfssl/
Daily Blog	https://wolfssl.com/wolfSSL/Blog/Blog.html

Chapter 17: wolfSSL (formerly CyaSSL) API Reference

17.1 Initialization / Shutdown

The functions in this section have to do with initializing the wolfSSL library and shutting it down (freeing resources) after it is no longer needed by the application.

wolfSSL_Init

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_Init(void);

Description:
Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_MUTEX_E is an error that may be returned.

WC_INIT_E wolfCrypt initialization error returned.

Parameters:

This function has no parameters.

Example:

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
      /*failed to initialize wolfSSL library*/
}
```

See Also:
wolfSSL_Cleanup

wolfSSL_library_init

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_library_init(void)

Description:
This function is called internally in wolfSSL_CTX_new().

This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_FATAL_ERROR is returned upon failure.

Parameters:

This function takes no parameters.

Example:

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
      /*failed to initialize wolfSSL*/
}
...
```

See Also:
wolfSSL_Init
wolfSSL_Cleanup

wolfSSL_Cleanup

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_Cleanup(void);

Description:

Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.

Return Values:

SSL_SUCCESS return no errors.

BAD_MUTEX_E a mutex error return.

Parameters:

There are no parameters for this function.

Example:

```
wolfSSL_Cleanup();
```

See Also:

wolfSSL_Init

wolfSSL_shutdown

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_shutdown(WOLFSSL* ssl);

Description:

This function shuts down an active SSL/TLS connection using the SSL session, **ssl**. This function will try to send a "close notify" alert to the peer.

The calling application can choose to wait for the peer to send its "close notify" alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two-directional shutdown procedure must be performed to keep synchronization intact between the peers.

wolfSSL_shutdown() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_shutdown() will return an error if the underlying I/O could not satisfy the needs of wolfSSL_shutdown() to continue. In this case, a call to wolfSSL_get_error() will yield either **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process must then repeat the call to wolfSSL_shutdown() when the underlying I/O is ready.

Return Values:

SSL_SUCCESS - will be returned upon success.

SSL_SHUTDOWN_NOT_DONE - will be returned when shutdown has not finished, and the function should be called again.

SSL_FATAL_ERROR - will be returned upon failure. Call wolfSSL_get_error() for a more specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
      /*failed to shut down SSL connection*/
}
```

See Also:

wolfSSL_free
wolfSSL_CTX_free

wolfSSL_get_shutdown

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_get_shutdown(WOLFSSL* ssl);

Description:
This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.

Return Values:
1 - SSL_SENT_SHUTDOWN is returned.

2 - SSL_RECEIVED_SHUTDOWN is returned.

Parameters:

ssl - a constant pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
int ret;
ret = wolfSSL_get_shutdown(ssl);

if(ret == 1){
      /*SSL_SENT_SHUTDOWN */
} else if(ret == 2){
      /*SSL_RECEIVED_SHUTDOWN */
} else {
      /*Fatal error.*/
}
```

See Also:
wolfSSL_SESSION_free

wolfSSL_is_init_finished

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_is_init_finished(WOLFSSL* ssl);

Description:
This function checks to see if the connection is established.

Return Values:
0 - returned if the connection is not established, i.e. the WOLFSSL struct is NULL or the handshake is not done.

1 - returned if the handshake is done.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
      /*Handshake is done and connection is established*/
}
```

See Also:
wolfSSL_set_accept_state
wolfSSL_get_keys
wolfSSL_set_shutdown

wolfSSL_ALPN_GetPeerProtocol

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_ALPN_GetPeerProtocol(WOLFSSL* ssl, char** list, word16* listSz);

Description:
This function copies the alpn_client_list data from the SSL object to the buffer.

Return Values:
SSL_SUCCESS - returned if the function executed without error. The alpn_client_list member of the SSL object has been copied to the **list** parameter.

BAD_FUNC_ARG - returned if the **list** or **listSz** parameter is NULL.

BUFFER_ERROR - returned if there will be a problem with the **list** buffer (either it's NULL or the size is 0).

MEMORY_ERROR - returned if there was a problem dynamically allocating memory.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

list - a pointer to the buffer. The data from the SSL object will be copied into it.

listSz - the buffer size.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#ifdef HAVE_ALPN
char* list = NULL;
word16 listSz = 0;
…
err = wolfSSL_ALPN_GetPeerProtocol(ssl, &list, &listSz);

if(err == SSL_SUCCESS){
      /*List of protocols names sent by client */
}
```

See Also:
wolfSSL_UseALPN

wolfSSL_SetMinVersion

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetMinVersion(WOLFSSL* ssl, int version);

Description:
This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Return Values:
SSL_SUCCESS - returned if this function and its subroutine executes without error.

BAD_FUNC_ARG - returned if the SSL object is NULL. In the subroutine this error is thrown if there is not a good version match.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

version - an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int version = /*version id (see internal.h enum Misc)*/
…
if(version != SSL_SUCCESS){
	/*The minimum version failed to set properly */
} else {
	/*You have successfully set the min version */
}
```

See Also:
SetMinVersionHelper
wolfSSL_CTX_SetMinVersion

wolfSSL_MakeTlsMasterSecret

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_MakeTlsMasterSecret(byte* ms, word32 msLen, const byte* pms,
word32 pmsLen, const byte* cr, const byte* sr, int tls1_2, int hash_type);

Description:
This function copies the values of **cr** and **sr** then passes through to PRF (pseudo random function) and returns that value.

Return Values:
This function returns **0** on success.

BUFFER_E - returned if there will be an error with the size of the buffer.

MEMORY_E - returned if a subroutine failed to allocate dynamic memory.

Parameters:

ms - the master secret held in the Arrays structure.

msLen - the length of the master secret.

pms - the pre-master secret held in the Arrays structure.

pmsLen - the length of the pre-master secret.

cr - the client random.

sr - the server random.

tls1_2 - signifies that the version is at least tls version 1.2.

hash_type - signifies the hash type.

Example:

```
WOLFSSL* ssl;  /*Initialize*/
```

```
/*called in MakeTlsMasterSecret and retrieves the necessary information as
follows:*/

int MakeTlsMasterSecret(WOLFSSL* ssl){
      int ret;
      ret = wolfSSL_makeTlsMasterSecret(ssl->arrays->masterSecret,
SECRET_LEN,
                        ssl->arrays->preMasterSecret, ssl->arrays-
                        >preMasterSz,
                        ssl->arrays->clientRandom, ssl->arrays->serverRandom,
                        IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);

…
return ret;

}
```

See Also:
PRF
doPRF
p_hash
MakeTlsMasterSecret

wolfSSL_SetServerID

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetServerID(WOLFSSL* ssl, const byte* id, int len, int newSession);

Description:
This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.

Return Values:
SSL_SUCCESS - returned if the function executed without an error.

BAD_FUNC_ARG - returned if the WOLFSSL struct or **id** parameter is NULL or if **len** is not greater than zero.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

id - a constant byte pointer that will be copied to the **serverID** member of the WOLFSSL_SESSION structure.

len - an int type representing the length of the session **id** parameter.

newSession - an int type representing the flag to denote whether to reuse a session or not.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE];  /*or dynamically create space*/
int len = 0; /*initialize length*/
int newSession = 0; /*flag to allow*/
…
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if(ret){
      /*The Id was successfully set*/
}
```

See Also:
GetSessionClient

wolfSSL_ALPN_GetProtocol

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_ALPN_GetProtocol(WOLFSSL* ssl, char** protocol_name, word16* size);

Description:
This function gets the protocol name set by the server.

Return Values:
SSL_SUCCESS - returned on successful execution where no errors were thrown.

SSL_FATAL_ERROR - returned if the extension was not found or if there was no protocol match with peer. There will also be an error thrown if there is more than one protocol name accepted.

SSL_ALPN_NOT_FOUND - returned signifying that no protocol match with peer was found.

BAD_FUNC_ARG - returned if there was a NULL argument passed into the function.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

protocol_name - a pointer to a char that represents the protocol name and will be held in the ALPN structure.

size - a word16 type that represents the size of the protocol_name.

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int err;
char* protocol_name = NULL;
Word16 protocol_nameSz = 0;
err = wolfSSL_ALPN_GetProtocol(ssl, &protocol_name, &protocol_nameSz);

if(err == SSL_SUCCESS){
      /*Sent ALPN protocol*/
}
```

See Also:
TLSX_ALPN_GetRequest
TLSX_Find

17.2 Certificates and Keys

The functions in this section have to do with loading certificates and keys into wolfSSL.

wolfSSL_CTX_load_verify_buffer

Synopsis:

int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,
long sz, int format);

Description:

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. **format** specifies the format type of the buffer; **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

in - pointer to the CA certificate buffer

sz - size of the input CA certificate buffer, **in**.

format - format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];

...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*error loading CA certs from buffer*/
}

...
```

See Also:
wolfSSL_CTX_load_verify_locations
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_buffer
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_CTX_load_verify_locations

Synopsis:
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file,
const char* path);

Description:
This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake.

The root certificate file, provided by the **file** argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The **path** argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of **file** is not NULL, **path** may be specified as NULL if not needed. If **path** is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory and locate any files with the PEM header "-----BEGIN CERTIFICATE-----".

Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_FAILURE will be returned if **ctx** is NULL, or if both **file** and **path** are NULL.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

BAD_PATH_ERROR will be returned if opendir() fails when trying to open **path**.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

file - pointer to name of the file containing PEM-formatted CA certificates

path - pointer to the name of a directory to load PEM-formatted certificates from.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", 0);
if (ret != SSL_SUCCESS) {
      /*error loading CA certs*/
}
```

. . .

See Also:
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_file
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_file
wolfSSL_use_certificate_file
wolfSSL_use_PrivateKey_file
wolfSSL_use_certificate_chain_file

wolfSSL_CTX_use_PrivateKey_buffer

Synopsis:
int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);

Description:
This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. **format** specifies the format type of the buffer; **SSL_FILETYPE_ASN1**or **SSL_FILETYPE_PEM**. Please see the examples for proper usage.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

in - the input buffer containing the private key to be loaded.

sz - the size of the input buffer.

format - the format of the private key located in the input buffer (**in**). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];

...

ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*error loading private key from buffer*/
}

...
```

See Also:
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_buffer
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_CTX_use_PrivateKey_file

Synopsis:
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX* ctx, const char* file,

int format);

Description:

This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the **file** argument. The **format** argument specifies the format type of the file - **SSL_FILETYPE_ASN1**or **SSL_FILETYPE_PEM**. Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The file is in the wrong format, or the wrong format has been given using the "format" argument
- The file doesn't exist, can't be read, or is corrupted
- An out of memory condition occurs
- Base16 decoding fails on the file
- The key file is encrypted but no password is provided

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_use_PrivateKey_file(ctx, "./server-key.pem",
                                    SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*error loading key file*/
}

...
```

See Also:

wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_use_PrivateKey_file
wolfSSL_use_PrivateKey_buffer

wolfSSL_get_privateKey

Synopsis:
WOLFSSL_EVP_PKEY *wolfSSL_get_privatekey(const WOLFSSL *ssl)

Description:
This function gets a pointer to a private-key of the X.509 certificate in the SSL.

Return Values:
If successful the call will return EVP_PKEY of the SSL, otherwise NULL will be returned when No private key is loaded and getting a private key failed.

Example:

```
WOLFSSL* ssl;
WOLFSSL_EVP_PKEY* evp_key;
...

evp_key = wolfSSL_get_privatekey(ssl);

...
```

See Also:
wolfSSL_CTX_new
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_new
wolfSSL_EVP_PKEY_free
wolfSSL_free
wolfSSL_CTX_free

wolfSSL_CTX_use_certificate_buffer

Synopsis:
int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,
long sz, int format);

Description:
This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. **format** specifies the

format type of the buffer; **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**. Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

in - the input buffer containing the certificate to be loaded.

sz - the size of the input buffer.

format - the format of the certificate located in the input buffer (**in**). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];

...

ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*error loading certificate from buffer*/
}

...
```

See Also:

wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_buffer
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_CTX_use_certificate_chain_buffer

Synopsis:
int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx,
const unsigned char* in, long sz);

Description:
This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. The buffer must be in **PEM** format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

in - the input buffer containing the PEM-formatted certificate chain to be loaded.

sz - the size of the input buffer.

Example:

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];

...

ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
      /*error loading certificate chain from buffer*/
}

...
```

See Also:
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_use_certificate_buffer
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_CTX_use_certificate_chain_file

Synopsis:
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX* ctx, const char* file);

Description:
This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the **file** argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The file is in the wrong format, or the wrong format has been given using the “format” argument
- file doesn’t exist, can’t be read, or is corrupted
- an out of memory condition occurs

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()

file - a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL context. Certificates must be in PEM format.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_use_certificate_chain_file(ctx, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
      /*error loading cert file*/
}

...
```

See Also:

wolfSSL_CTX_use_certificate_file
wolfSSL_CTX_use_certificate_buffer
wolfSSL_use_certificate_file
wolfSSL_use_certificate_buffer

wolfSSL_CTX_use_certificate_file

Synopsis:

int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX* ctx, const char* file, int format);

Description:

This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the **file** argument. The **format** argument specifies the format type of the file, either **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**. Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The file is in the wrong format, or the wrong format has been given using the "format" argument
- file doesn't exist, can't be read, or is corrupted
- an out of memory condition occurs
- Base16 decoding fails on the file

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()

file - a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL context.

format - format of the certificates pointed to by **file**. Possible options are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_use_certificate_file(ctx, "./client-cert.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*error loading cert file*/
}

...
```

See Also:

wolfSSL_CTX_use_certificate_buffer
wolfSSL_use_certificate_file
wolfSSL_use_certificate_buffer

wolfSSL_SetTmpDH

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p, int pSz, unsigned char* g,
int gSz);

Description:

Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.

Return Values:

If successful the call will return **SSL_SUCCESS**.

MEMORY_ERROR will be returned if a memory error was encountered.

SIDE_ERROR will be returned if this function is called on an SSL client instead of an SSL server.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

p - Diffie-Hellman prime number parameter.

pSz - size of **p**.

g - Diffie-Hellman “generator” parameter.

gSz - size of **g**.

Example:

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
```

```
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

See Also:

SSL_accept

wolfSSL_use_PrivateKey

Synopsis:

#include <wolfssl/ssl.h>

SSL_use_PrivateKey ->

int wolfSSL_use_PrivateKey(WOLFSSL* ssl, WOLFSSL_EVP_PKEY* pkey);

Description:

This is used to set the private key for the WOLFSSL structure.

Return Values:

SSL_SUCCESS: On successful setting argument.

SSL_FAILURE: If an NULL ssl passed in.

All error cases will be negative values.

Parameters:

ssl - WOLFSSL structure to set argument in.

pkey - private key to use.

Example:

```
WOLFSSL* ssl;

WOLFSSL_EVP_PKEY* pkey;

int ret;

// create ssl object and set up private key

ret  = wolfSSL_use_PrivateKey(ssl, pkey);
// check ret value
```

See Also:
wolfSSL_new, wolfSSL_free, wolfSSL_use_PrivateKey

wolfSSL_use_PrivateKey_ASN1

Synopsis:
#include <wolfssl/ssl.h>

SSL_use_PrivateKey_ASN1 ->
int wolfSSL_use_PrivateKey_ASN1(int pri, WOLFSSL* ssl, unsigned char* der, long derSz);

Description:
This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected

Return Values:
SSL_SUCCESS: On successful setting parsing and setting the private key.

SSL_FAILURE: If an NULL ssl passed in.

All error cases will be negative values.

Parameters:
pri - type of private key.

ssl - WOLFSSL structure to set argument in.

der -buffer holding DER key.

derSz - size of der buffer.

Example:
```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
```

```
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free, wolfSSL_use_PrivateKey

wolfSSL_use_RSAPrivateKey_ASN1

Synopsis:

#include <wolfssl/ssl.h>

SSL_use_RSAPrivateKey_ASN1 ->
int wolfSSL_use_RSAPrivateKey_ASN1(WOLFSSL* ssl, unsigned char* der, long derSz);

Description:

This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected

Return Values:

SSL_SUCCESS: On successful setting parsing and setting the private key.

SSL_FAILURE: If an NULL ssl passed in.

All error cases will be negative values.

Parameters:

ssl - WOLFSSL structure to set argument in.

der -buffer holding DER key.

derSz - size of der buffer.

Example:

```
WOLFSSL* ssl;
unsigned char* pkey;
```

```
long pkeySz;
int ret;
// create ssl object and set up RSA private key
ret  = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free, wolfSSL_use_PrivateKey

wolfSSL_use_PrivateKey_buffer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_use_PrivateKey_buffer(WOLFSSL* ssl, const unsigned char* in,
long sz, int format);

Description:

This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. **format** specifies the format type of the buffer; **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**. Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

in - buffer containing private key to load.

sz - size of the private key located in **buffer**.

format - format of the private key to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int buffSz;
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
...

ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*failed to load private key from buffer*/
}
```

See Also:
wolfSSL_use_PrivateKey
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_use_certificate_buffer

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_use_certificate_buffer(WOLFSSL* ssl, const unsigned char* in,
long sz, int format);

Description:
This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. **format** specifies the format type of the buffer; **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**. Please see the examples for proper usage.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

in - buffer containing certificate to load.

sz - size of the certificate located in **buffer**.

format - format of the certificate to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
Example:

```
int buffSz;
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
...

ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      /*failed to load certificate from buffer*/
}
```

See Also:
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_use_certificate_chain_buffer

Synopsis:
```
#include <wolfssl/ssl.h>

int wolfSSL_use_certificate_chain_buffer(WOLFSSL* ssl, const unsigned char* in,
                                         long sz);
```

Description:
This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **sz**. The buffer must be in **PEM** format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

in - buffer containing certificate to load.

sz - size of the certificate located in **buffer**.

Example:

```
int buffSz;
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
...

ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
      /*failed to load certificate chain from buffer*/
}
```

See Also:
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_buffer
wolfSSL_use_PrivateKey_buffer

wolfSSL_CTX_der_load_verify_locations

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_der_load_verify_locations(WOLFSSL_CTX* ctx, const char* file,
int format);

Description:
This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER-formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM-formatted CA files as well. These certificates will be treated as trusted root

certificates and used to verify certs received from peers during the SSL handshake.

The root certificate file, provided by the **file** argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The **format** argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path.

Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned upon failure.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()

file - a pointer to the name of the file containing the CA certificates to be loaded into the wolfSSL SSL context, with format as specified by **format**.

format - the encoding type of the certificates specified by **file**. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_der_load_verify_locations(ctx, “./ca-cert.der”,
                                            SSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
      /*error loading CA certs*/
}

...
```

See Also:

wolfSSL_CTX_load_verify_locations
wolfSSL_CTX_load_verify_buffer

wolfSSL_CTX_use_NTRUPrivateKey_file

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_use_NTRUPrivateKey_file(WOLFSSL_CTX* ctx, const char* file);

Description:
This function loads an NTRU private key file into the WOLFSSL Context. It behaves like the normal version, only differing in its ability to accept an NTRU raw key file. This function is needed since the format of the file is different than the normal key file (buffer) functions. Please see the examples for proper usage.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()

file - a pointer to the name of the file containing the NTRU private key to be loaded into the wolfSSL SSL context.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_use_NTRUPrivateKey_file(ctx, "./ntru-key.raw");
if (ret != SSL_SUCCESS) {
      /*error loading NTRU private key*/
}

...
```

See Also:
wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_buffer
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_KeepArrays

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_KeepArrays(WOLFSSL* ssl);

Description:
Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints.

When the user is done with temporary arrays, either **wolfSSL_FreeArrays()** may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.

Return Values:

This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

See Also:

wolfSSL_FreeArrays

wolfSSL_FreeArrays

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_FreeArrays(WOLFSSL* ssl);

Description:

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.

Return Values:

This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl;
```

```
...
wolfSSL_FreeArrays(ssl);
```

See Also:
wolfSSL_KeepArrays

wolfSSL_UnloadCertsKeys

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_UnloadCertsKeys(WOLFSSL* ssl);

Description:
This function unloads any certificates or keys that SSL owns.

Return Values:
SSL_SUCCESS - returned if the function executed successfully.

BAD_FUNC_ARG - returned if the WOLFSSL object is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:
wolfSSL_CTX_UnloadCAs

wolfSSL_CTX_get_cert_cache_memsize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_get_cert_cache_memsize(WOLFSSL_CTX* ctx);

Description:
Returns the size the certificate cache save buffer needs to be.

Return Values:
If the funciton is successful an **INTEGER** value is returned representing the memory size.

BAD_FUNC_ARG is returned if the WOLFSSL_CTX struct is NULL.

BAD_MUTEX_E - returned if there was a mutex lock error.

Parameters:

ctx - a pointer to a wolfSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol*/);
…
int certCacheSize = wolfSSL_CTX_get_cert_cache_memsize(ctx);

if(certCacheSize != BAD_FUNC_ARG || certCacheSize != BAD_MUTEX_E){
      /*Successfully retrieved the memory size. */
}
```

See Also:
CM_GetCertCacheMemSize

wolfSSL_X509_get_signature_type

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_X509_get_signature_type(WOLFSSL_X509* x509);

Description:
This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.

Return Values:
0 - returned if the WOLFSSL_X509 structure is NULL.

An **Integer** value is returned which was retrieved from the x509 object.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                          DYNAMIC_TYPE_X509);

…
int x509SigType = wolfSSL_X509_get_signature_type(x509);

if(x509SigType != EXPECTED){
      /*Deal with an unexpected value*/
}
```

See Also:
wolfSSL_X509_get_signature
wolfSSL_X509_version
wolfSSL_X509_get_der
wolfSSL_X509_get_serial_number
wolfSSL_X509_notBefore
wolfSSL_X509_notAfter
wolfSSL_X509_free

wolfSSL_X509_get_next_altname

Synopsis:
#include <wolfssl/ssl.h>

char* wolfSSL_X509_get_next_altname(WOLFSSL_X509* cert);

Description:
This function returns the next, if any, altname from the peer certificate.

Return Values:
NULL if there is not a next altname.

cert->altNamesNext->name from the WOLFSSL_X509 structure that is a string value from the altName list is returned if it exists.

Parameters:

cert - a pointer to the wolfSSL_X509 structure.

Example:

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                            DYNAMIC_TYPE_X509);

…
int x509NextAltName = wolfSSL_X509_get_next_altname(x509);

if(x509NextAltName == NULL){
      /*There isn't another alt name*/
}
```

See Also:
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_subject_name

wolfSSL_X509_get_subjectCN

Synopsis:
#include <wolfssl/ssl.h>

char* wolfSSL_X509_get_subjectCN(WOLFSSL_X509* x509);

Description:
Returns the common name of the subject from the certificate.

Return Values:

NULL - returned if the x509 structure is null

A **string** representation of the subject's common name is returned if the function executes successfully.

Parameters:

x509 - a pointer to a WOLFSSL_X509 structure containing certificate information.

Example:

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                          DYNAMIC_TYPE_X509);

…
int x509Cn = wolfSSL_X509_get_subjectCN(x509);

if(x509Cn == NULL){
      /*Deal with NULL case*/
} else {
      /*x509Cn contains the common name*/
}
```

See Also:

wolfSSL_X509_Name_get_entry
wolfSSL_X509_get_next_altname
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_subject_name

wolfSSL_X509_get_der

Synopsis:

#include <wolfssl/ssl.h>

const byte* wolfSSL_X509_get_der(WOLFSSL_X509* x509, int* outSz);

Description:

This function gets the DER encoded certificate in the WOLFSSL_X509 struct.

Return Values:

This function returns the DerBuffer structure's **buffer** member, which is of type byte.

NULL - returned if the **x509** or **outSz** parameter is NULL.

Parameters:

x509 - a pointer to a WOLFSSL_X509 structure containing certificate information.

outSz - length of the derBuffer member of the WOLFSSL_X509 struct.

Example:

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                          DYNAMIC_TYPE_X509);
int* outSz; /*initialize*/
…
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);

if(x509Der == NULL){
      /*Failure case one of the parameters was NULL */
}
```

See Also:
wolfSSL_X509_version
wolfSSL_X509_Name_get_entry
wolfSSL_X509_get_next_altname
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_subject_name

wolfSSL_X509_get_hw_type

Synopsis:
#include <wolfssl/ssl.h>

byte* wolfSSL_X509_get_hw_type(WOLFSSL_X509* x509, byte* in, int* inOutSz);

Description:
The function copies the **hwType** member of the WOLFSSL_X509 structure to the buffer.

Return Values:

The function returns a **byte type** of the data previously held in the **hwType** member of the WOLFSSL_X509 structure.

NULL - returned if **inOutSz** is NULL.

Parameters:

x509 - a pointer to a WOLFSSL_X509 structure containing certificate information.

in - pointer to type byte that represents the buffer.

inOutSz - pointer to type int that represents the size of the buffer.

Example:

```
WOLFSSL_X509* x509;  /*X509 certificate*/
byte* in;  /*initialize the buffer*/
int* inOutSz;  /*holds the size of the buffer*/
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);

if(hwType == NULL){
      /*Failure case function returned NULL. */
}
```

See Also:
wolfSSL_X509_get_hw_serial_number
wolfSSL_X509_get_device_type

wolfSSL_X509_d2i_fp

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_X509* wolfSSL_X509_d2i_fp(WOLFSSL_X509** x509, XFILE file);

Description:
If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.

Return Values:

WOLFSSL_X509 structure pointer is returned if the function executes successfully.

NULL - if the call to XFTELL macro returns a negative value.

Parameters:

x509 - a pointer to a WOLFSSL_X509 pointer.

file - a defined type that is a pointer to a FILE.

Example:

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file;  (mapped to struct fs_file*)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);

if(newX509 == NULL){
      /*The function returned NULL */
}
```

See Also:

wolfSSL_X509_d2i
XFTELL
XREWIND
XFSEEK

wolfSSL_SetCertCbCtx

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_SetCertCbCtx(WOLFSSL* ssl, void* ctx);

Description:

This function stores user CTX object information for verify callback.

Return Values:

This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

ctx - a void pointer that is set to WOLFSSL structure's verifyCbCtx member's value.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
(void*)ctx;
…
if(ssl != NULL){
      wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
      /*Error case, the SSL is not initialized properly. */
}
```

See Also:

wolfSSL_CTX_save_cert_cache
wolfSSL_CTX_restore_cert_cache
wolfSSL_CTX_set_verify

wolfSSL_CertPemToDer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CertPemToDer(const unsigned char* pem, int pemSz,
Unsigned char* buff, int buffSz, int type);

Description:

This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.

Return Values:
Returns the bytes written to the buffer.

Parameters:

pem - pointer PEM formatted certificate.

pemSz - size of the certificate.

buff - buffer to be copied to DER format.

buffSz - size of the buffer.

type - Certificate file type found in asn_public.h **enum CertType.**

Example:

```
const unsigned char* pem;
int pemSz;
unsigned char buff[BUFSIZE];
int buffSz = sizeof(buff)/sizeof(char);
int type;
...
if(wolfSSL_CertPemToDer(pem, pemSz, buff, buffSz, type) <= 0) {
	/*There were bytes written to buffer*/
}
```

See Also:
PemToDer (OpenSSL)

wolfSSL_X509_notAfter

Synopsis:
#include <wolfssl/ssl.h>

const byte* wolfSSL_X509_notAfter(wolfSSL_X509* x509);

Description:
This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.

Return Values:
The function returns a **constant byte pointer** to the notAfter member of the x509 struct.

NULL - returned if the x509 object is NULL.

Parameters:

x509 - a pointer to the WOLFSSL_X509 struct.

Example:

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
                                    DYNAMIC_TYPE_X509) ;
...
byte* notAfter = wolfSSL_X509_notAfter(x509);
if(notAfter == NULL){
      /*Failure case, the x509 object is null. */
}
```

See Also:
wolfssl/openssl/ssl.h
cyassl/ssl.h

wolfSSL_get_peer_certificate

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_X509* wolfSSL_get_peer_certificate(WOLFSSL* ssl);

Description:
This function gets the peer's certificate.

Return Values:
Returns a **pointer** to the peerCert member of the WOLFSSL_X509 structure if it exists.

0 - returned if the peer certificate issuer size is not defined.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
      /*You have a pointer peerCert to the peer certification*/
}
```

See Also:
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_subject_name
wolfSSL_X509_get_isCA

wolfSSL_get_peer_cert_chain

Synopsis:
#include <wolfssl/ssl.h>

STACK_OF(WOLFSSL_X509)* wolfSSL_get_peer_cert_chain(const WOLFSSL* ssl);

Description:
This function gets the peer's certificate chain.

Return Values:
Returns a **pointer** to the peer's Certificate stack.

NULL - returned if no peer certificate.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
         wolfSSL_connect(ssl);

STACK_OF(WOLFSSL_X509)* chain = wolfSSL_get_peer_cert_chain(ssl);

ifchain){
      /*You have a pointer to the peer certificate chain*/
}
```

See Also:
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_subject_name
wolfSSL_X509_get_isCA

wolfSSL_X509_get_isCA

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_X509_get_isCA(WOLFSSL_X509* x509);

Description:
Checks the isCa member of the WOLFSSL_X509 structure and returns the value.

Return Values:
The value in the **isCA member** of the WOLFSSL_X509 structure is returned.

0 - returned if there is not a valid x509 structure passed in.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl;
```

```
…
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(wolfSSL_X509_get_isCA(ssl)){
	/*This is the CA*/
}else {
	/*Failure case*/
}
```

See Also:
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_isCA

wolfSSL_CTX_save_cert_cache

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_save_cert_cache(WOLFSSL_CTX* ctx, const char* fname);

Description:
This function writes the cert cache from memory to file.

Return Values:
SSL_SUCCESS - if CM_SaveCertCache exits normally.

BAD_FUNC_ARG - is returned if either of the arguments are NULL.

SSL_BAD_FILE - if the cert cache save file could not be opened.

BAD_MUTEX_E - if the lock mutex failed.

MEMORY_E - the allocation of memory failed.

FWRITE_ERROR - Certificate cache file write failed.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, holding the certificate information.

fname - the cert cache buffer.

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol def*/);
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
      /*file was written. */
}
```

See Also:
CM_SaveCertCache
DoMemSaveCertCache

wolfSSL_CTX_restore_cert_cache

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_restore_cert_cache(WOLFSSL_CTX* ctx,
const char* fname) ;

Description:
This function persistes certificate cache from a file.

Return Values:
SSL_SUCCESS - returned if the function, CM_RestoreCertCache, executes normally.

SSL_BAD_FILE - returned if XFOPEN returns XBADFILE. The file is corrupted.

MEMORY_E - returned if the allocated memory for the temp buffer fails.

BAD_FUNC_ARG - returned if fname or ctx have a NULL value.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, holding the certificate information.

fname - the cert cache buffer.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = /*path to file*/;
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
      /*check to see if the execution was successful */
}
```

See Also:
CM_RestoreCertCache
XFOPEN

wolfSSL_get_chain_X509

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_X509*
wolfSSL_get_chain_X509(WOLFSSL_X509_CHAIN* chain, int idx);

Description:
This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.

Return Values:
The function returns a pointer to a WOLFSSL_X509 structure.

Parameters:

chain - a pointer to the WOLFSSL_X509_CHAIN used for no dynamic memory SESSION_CACHE.

idx - the index of the WOLFSSL_X509 certificate.

Example:

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = /*set idx*/;
…
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
/*ptr contains the cert at the index specified*/
} else {
      /*ptr is NULL*/
}
```

See Also:

InitDecodedCert
ParseCertRelative
CopyDecodedToX509

wolfSSL_wolfSSL_X509_notBefore

Synopsis:

#include <wolfssl/ssl.h>

const byte* wolfSSL_X509_notBefore(WOLFSSL_X509* x509);

Description:

The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

Return Values:

This function returns a **constant byte pointer** to the x509's member notAfter.

NULL - the function returns NULL if the x509 structure is NULL.

Parameters:

x509 - a pointer to the WOLFSSL_X509 struct.

Example:

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                              DYNAMIC_TYPE_X509) ;
…
byte* notAfter = wolfSSL_X509_notAfter(x509);
```

```
if(notAfter == NULL){
	/*The x509 object was NULL */
}
```

See Also:
wolfSSL_X509_notAfter

wolfSSL_X509_get_signature

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_X509_get_signature(WOLFSSL_X509* x509,
unsigned char* buf, int bufSz);

Description:
Gets the X509 signature and stores it in the buffer.

Return Values:
SSL_SUCCESS - returned if the function successfully executes. The signature is loaded into the buffer.

SSL_FATAL_ERRROR - returns if the x509 struct or the bufSz member is NULL. There is also a check for the length member of the sig structure (sig is a member of x509).

Parameters:

x509 - pointer to a WOLFSSL_X509 structure..

buf - a char pointer to the buffer.

bufSz - an integer pointer to the size of the buffer.

Example:

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                          DYNAMIC_TYPE_X509);
```

```
unsigned char* buf; /*Initialize*/
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){
	/*The function did not execute successfully. */
} else{
	/*The buffer was written to correctly. */
}
```

See Also:
wolfSSL_X509_get_serial_number
wolfSSL_X509_get_signature_type
wolfSSL_X509_get_device_type

wolfSSL_X509_get_device_type

Synopsis:
#include <wolfssl/ssl.h>

byte* wolfSSL_X509_get_device_type(WOLFSSL_X509* x509, byte* in,
int* inOutSz);

Description:
This function copies the device type from the x509 structure to the buffer.

Return Values:
Returns a **byte pointer** holding the device type from the x509 structure.

NULL - returned if the buffer size is NULL.

Parameters:

x509 - pointer to a WOLFSSL_X509 structure, created with WOLFSSL_X509_new().

in - a pointer to a byte type that will hold the device type (the buffer).

inOutSz - the minimum of either the parameter inOutSz or the deviceTypeSz member of the x509 structure.

Example:

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                            DYNAMIC_TYPE_X509);
byte* in;
int* inOutSz;
...
byte* deviceType = wolfSSL_X509_get_device_type(x509, in, inOutSz);

if(!deviceType){
      /*Failure case, NULL was returned. */
}
```

See Also:
wolfSSL_X509_get_hw_type
wolfSSL_X509_get_hw_serial_number
wolfSSL_X509_d2i

wolfSSL_CTX_memsave_cert_cache

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_memsave_cert_cache(WOLFSSL_CTX* ctx, void* mem,
int sz, int* used);

Description:
This function persists the certificate cache to memory.

Return Values:
SSL_SUCCESS - returned on successful execution of the function. No errors were thrown.

BAD_MUTEX_E - mutex error where the WOLFSSL_CERT_MANAGER member caLock was not 0 (zero).

BAD_FUNC_ARG - returned if **ctx**, **mem**, or **used** is NULL or if **sz** is less than or equal to 0 (zero).

BUFFER_E - output buffer **mem** was too small.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

mem - a void pointer to the destination (output buffer).

sz - the size of the output buffer.

used - a pointer to size of the cert cache header.

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol*/);
void* mem;  /*initialize */
int sz; /*sizeof mem*/
int* used; /*cert cache header*/
...
if(wolfSSL_CTX_memsave_cert_cache(ctx, mem, sz, used) != SSL_SUCCESS){
      /*The function returned with an error*/
}
```

See Also:

DoMemSaveCertCache
GetCertCacheMemSize
CM_MemRestoreCertCache
CM_GetCertCacheMemSize

wolfSSL_KeyPemToDer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_KeyPemToDer(const unsigned char* pem, int pemSz,
                        unsigned char* buff, int buffSz, const char* pass);

Description:

Converts a key in PEM format to DER format.

Return Values:

The function returns the number of **bytes** written to the buffer on successful execution.

< 0 returned indicating an error.

Parameters:

pem - a pointer to the PEM encoded certificate.

pemSz - the size of the PEM buffer (**pem**).

buff - a pointer to the copy of the buffer member of the DerBuffer struct.

buffSz - size of the buffer space allocated in the DerBuffer struct.

pass - password passed into the function.

Example:

```
byte* loadBuf; /*Initialize */
long fileSz = 0;
byte* bufSz; /*Initialize */
static int LoadKeyFile(byte** keyBuf, word32* keyBufSz, const char* keyFile,
                              int typeKey, const char* pasword);
…
bufSz = wolfSSL_KeyPemToDer(loadBuf, (int)fileSz, saveBuf,
                              (int)fileSz, password);

if(saveBufSz > 0){
      /*Bytes were written to the buffer. */
}
```

See Also:
PemToDer
wolfssl_decrypt_buffer_key

wolfSSL_X509_load_certificate_file

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_X509* wolfSSL_X509_load_certificate_file(const char* fname, int format);

Description:
The function loads the x509 certificate into memory.

Return Values:
A successful execution returns **pointer** to a WOLFSSL_X509 structure.

NULL - returned if the certificate was not able to be written.

Parameters:

fname - the certificate file to be loaded.

format - the format of the certificate.

Example:

```
#define cliCert    "certs/client-cert.pem"
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

See Also:
InitDecodedCert
PemToDer
wolfSSL_get_certificate
AssertNotNull

wolfSSL_X509_get_issuer_name

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_X509_NAME* wolfSSL_X509_get_issuer_name(WOLFSSL_X509* cert);

Description:
This function returns the name of the certificate issuer.

Return Values:
A **pointer** to the WOLFSSL_X509 struct's issuer member is returned.

NULL - if the cert passed in is NULL.

Parameters:

cert - a pointer to a WOLFSSL_X509 structure.

Example:

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
      /*NULL was returned*/
} else {
      /*issuer hods the name of the certificate issuer. */
}
```

See Also:
wolfSSL_X509_get_subject_name
wolfSSL_X509_get_isCA
wolfSSL_get_peer_certificate
wolfSSL_X509_NAME_oneline

wolfSSL_X509_NAME_oneline

Synopsis:
#include <wolfssl/ssl.h>

char* wolfSSL_X509_NAME_oneline(WOLFSSL_X509* name, char* in, int sz);

Description:
This function copies the name of the x509 into a buffer.

Return Values:
A **char pointer** to the buffer with the WOLFSSL_X509_NAME structures name member's data is returned if the function executed normally.

Parameters:
name - a pointer to a WOLFSSL_X509 structure.

in - a buffer to hold the name copied from the WOLFSSL_X509_NAME structure.

sz - the maximum size of the buffer.

Example:

```
WOLFSSL_X509 x509; /*Initialize */
char* name;
...
name = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(name <= 0){
      /*There's nothing in the buffer. */
}
```

See Also:
wolfSSL_X509_get_subject_name
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_isCA
wolfSSL_get_peer_certificate
wolfSSL_X509_version

wolfSSL_X509_get_hw_serial_number

Synopsis:
#include <wolfssl/ssl.h>

byte* wolfSSL_X509_get_hw_serial_number(WOLFSSL_X509 x509, byte* in,
int* inOutSz);

Description:
This function returns the hwSerialNum member of the x509 object.

Return Values:
The function returns a **byte pointer** to the in buffer that will contain the serial number loaded from the x509 object.

Parameters:

x509 - pointer to a WOLFSSL_X509 structure containing certificate information.

in - a pointer to the buffer that will be copied to.

inOutSz - a pointer to the size of the buffer.

Example:

```
char* serial;
byte* in;  /*Initialize*/
int* inOutSz; /*Initialize to max size of buffer*/
WOLFSSL_X509 x509;
...
serial = wolfSSL_X509_get_hw_serial_number(x509, in, inOutSz);

if(serial == NULL || serial <= 0){
      /*Failure case */
}
```

See Also:
wolfSSL_X509_get_subject_name
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_isCA
wolfSSL_get_peer_certificate
wolfSSL_X509_version

wolfSSL_X509_get_subject_name

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_X509_NAME* wolfSSL_X509_get_subject_name(WOLFSSL_X509* cert);

Description:
This function returns the **subject** member of the WOLFSSL_X509 structure.

Return Values:
A **pointer** to the WOLFSSL_X509_NAME structure. The pointer may be **NULL** if the

WOLFSSL_X509 struct is NULL or if the **subject** member of the structure is NULL.

Parameters:

cert - a pointer to a WOLFSSL_X509 structure.

Example:

```
WOLFSSL_X509* cert;  /* Will be initialized */
WOLFSSL_X509_NAME name;
…
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
      /*Deal with the NULL case */
}
```

See Also:
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_isCA
wolfSSL_get_peer_certificate

wolfSSL_X509_version

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_X509_version(WOLFSSL_X509* x509);

Description:
This function retrieves the version of the X509 certificate.

Return Values:
0 - returned if the x509 structure is NULL.

The **version** stored in the x509 structure will be returned.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_X509* x509;  /*Initialize */
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
      /*The function returned 0, failure case. */
}
```

See Also:
wolfSSL_X509_get_subject_name
wolfSSL_X509_get_issuer_name
wolfSSL_X509_get_isCA
wolfSSL_get_peer_certificate

wolfSSL_DeriveTlsKeys

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_DeriveTlsKeys(byte* key_data, word32 keyLen, const byte* ms,
word32 msLen, const byte* sr, const byte* cr,
int tls1_2, int hash_type);

Description:
An external facing wrapper to derive TLS Keys.

Return Values:
0 - returned on success.

BUFFER_E - returned if the sum of **labLen** and **seedLen** (computes total size) exceeds the maximum size.

MEMORY_E - returned if the allocation of memory failed.

Parameters:

key_data - a byte pointer that is allocateded in DeriveTlsKeys and passed through to

PRF to hold the final hash.

keyLen - a word32 type that is derived in DeriveTlsKeys from the WOLFSSL structure's specs member.

ms - a constant pointer type holding the master secret held in the **arrays** structure within the WOLFSSL structure.

msLen - a word32 type that holds the length of the master secret in an enumerated define, SECRET_LEN.

sr - a constant byte pointer to the serverRandom member of the **arrays** structure within the WOLFSSL structure.

cr - a constant byte pointer to the clientRandom member of the **arrays** structure within the WOLFSSL structure.

tls1_2 - an integer type returned from IsAtLeastTLSv1_2().

hash_type - an integer type held in the WOLFSSL structure.

Example:

```
int DeriveTlsKeys(WOLFSSL* ssl){
int ret;
…
ret = wolfSSL_DeriveTlsKeys(key_data, length, ssl->arrays->masterSecret,
                  SECRET_LEN, ssl->arrays->clientRandom,
                  IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
}
```

See Also:
PRF
doPRF
DeriveTlsKeys
IsAtLeastTLSv1_2

wolfSSL_get_psk_identity

Synopsis:
#include <wolfssl/ssl.h>

const char* wolfSSL_get_psk_identity(const WOLFSSL* ssl);

Description:
The function returns a constant pointer to the client_identity member of the Arrays structure.

Return Values:
The **string** value of the client_identity member of the Arrays structure.

NULL - if the WOLFSSL structure is NULL or if the Arrays member of the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* pskID;
...
pskID = wolfSSL_get_psk_identity(ssl);

if(pskID == NULL){
      /*There is not a value in pskID*/
}
```

See Also:
wolfSSL_get_psk_identity_hint
wolfSSL_use_psk_identity_hint

wolfSSL_SetMinEccKey_Sz

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetMinEccKey_Sz(WOLFSSL* ssl, short keySz);

Description:

Sets the value of the minEccKeySz member of the **options** structure. The **options** struct is a member of the WOLFSSL structure and is accessed through the **ssl** parameter.

Return Values:

SSL_SUCCESS - if the function successfully set the minEccKeySz member of the **options** structure.

BAD_FUNC_ARG - if the WOLFSSL_CTX structure is NULL or if the key size (keySz) is less than 0 (zero) or not divisible by 8.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

keySz - value used to set the minimum ECC key size. Sets value in the **options** structure.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx); /*New session */
short keySz = /*min key size allowable*/;
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
      /*Failure case.  */

}
```

See Also:

wolfSSL_CTX_SetMinEccKey_Sz
wolfSSL_CTX_SetMinRsaKey_Sz
wolfSSL_SetMinRsaKey_Sz

wolfSSL_UseClientQSHKeys

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseClientQSHKeys(WOLFSSL* ssl, unsigned char flag);

Description:

If the flag is **1** keys will be sent in hello. If flag is **0** then the keys will not be sent during hello.

Return Values:
0 - on success.

BAD_FUNC_ARG - if the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().
flag - an unsigned char input to determine if the keys will be sent during hello.

Example:

```
WOLFSSL* ssl;
unsigned char flag = 1;  /*send keys*/
...
if(!wolfSSL_UseClientQSHKeys(ssl, flag)){
      /*The keys will be sent during hello. */
}
```

See Also:
wolfSSL_UseALPN
wolfSSL_UseSupportedQSH
wolfSSL_isQSH

wolfSSL_CTX_SetMinDhKey_Sz

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetMinDhKey_Sz(WOLFSSL_CTX* ctx, word16 keySz);

Description:
This function sets the minimum size of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.

Return Values:

SSL_SUCCESS - returned if the function completes successfully.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX struct is NULL or if the keySz is greater than 16,000 or not divisible by 8.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

keySz - a word16 type used to set the minimum DH key size. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

Example:

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
…
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKey);
```

See Also:
wolfSSL_SetMinDhKey_Sz
CTX_SetMinDhKey_Sz
wolfSSL_GetDhKey_Sz
wolfSSL_CTX_SetTMpDH_file

wolfSSL_CTX_SetTmpDH_buffer

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetTmpDH_buffer(WOLFSSL_CTX* ctx, const unsigned char* buf,
long sz, int format);

Description:
A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper

Return Values:
0 - returned for a successful execution.

BAD_FUNC_ARG - returned if the **ctx** or **buf** parameters are NULL.

MEMORY_E - if there is a memory allocation error.

SSL_BAD_FILETYPE - returned if **format** is not correct.

Parameters:

ctx - a pointer to a WOLFSSL structure, created using wolfSSL_CTX_new().

buf - a pointer to a constant unsigned char type that is allocated as the buffer and passed through to wolfSSL_SetTmpDH_buffer_wrapper.

sz - a long integer type that is derived from the **fname** parameter in wolfSSL_SetTmpDH_file_wrapper().

format - an integer type passed through from wolfSSL_SetTmpDH_file_wrapper().

Example:

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
                              Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; /*force heap usage*/
#else
byte* staticBuffer; /*Initialize */
long sz = 0;
…
if(ssl){
      ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
} else {
      ret = wolfSSL_CTX_SetTmpDH_buffer(ctx, myBuffer, sz, format);
}
```

See Also:

wolfSSL_SetTmpDH_buffer_wrapper
wolfSSL_SetTMpDH_buffer
wolfSSL_SetTmpDH_file_wrapper
wolfSSL_CTX_SetTmpDH_file

wolfSSL_GetIVSize

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_GetIVSize(WOLFSSL* ssl);

Description:

Returns the iv_size member of the **specs** structure held in the WOLFSSL struct.

Return Values:

Returns the value held in **ssl->specs.iv_size**.

BAD_FUNC_ARG - returned if the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int ivSize;
...
ivSize = wolfSSL_GetIVSize(ssl);

if(ivSize > 0){
      /*ivSize holds the specs.iv_size value. */
}
```

See Also:

wolfSSL_GetKeySize
wolfSSL_GetClientWriteIV
wolfSSL_GetServerWriteIV

wolfSSL_GetDhKey_Sz

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_GetDhKey_Sz(WOLFSSL* ssl);

Description:

Returns the value of dhKeySz that is a member of the **options** structure. This value represents the Diffie-Hellman key size in bytes.

Return Values:

Returns the value held in **ssl->options.dhKeySz** which is an integer value.

BAD_FUNC_ARG - returns if the WOLFSSL struct is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int dhKeySz;
...
dhKeySz = wolfSSL_GetDhKey_Sz(ssl);

if(dhKeySz == BAD_FUNC_ARG || dhKeySz <= 0){
      /*Failure case */
} else {
      /*dhKeySz holds the size of the key. */
}
```

See Also:

wolfSSL_SetMinDhKey_sz
wolfSSL_CTX_SetMinDhKey_Sz
wolfSSL_CTX_SetTmpDH
wolfSSL_SetTmpDH
wolfSSL_CTX_SetTmpDH_file

wolfSSL_SetTmpDH_buffer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_SetTmpDH_buffer(WOLFSSL* ssl, const unsigned char* buf,
long sz, int format);

Description:
The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.

Return Values:
SSL_SUCCESS - on successful execution.

SSL_BAD_FILETYPE - if the file type is not PEM and is not ASN.1. It will also be returned if the wc_DhParamsLoad does not return normally.

SSL_NO_PEM_HEADER - returns from PemToDer if there is not a PEM header.

SSL_BAD_FILE - returned if there is a file error in PemToDer.

SSL_FATAL_ERROR - returned from PemToDer if there was a copy error.

MEMORY_E - if there was a memory allocation error.

BAD_FUNC_ARG - returned if the WOLFSSL struct is NULL or if there was otherwise a NULL argument passed to a subroutine.

DH_KEY_SIZE_E - is returned if their is a key size error in wolfSSL_SetTmpDH() or in wolfSSL_CTX_SetTmpDH().

SIDE_ERROR - returned if it is not the server side in wolfSSL_SetTmpDH.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - allocated buffer passed in from wolfSSL_SetTMpDH_file_wrapper.

sz - a long int that holds the size of the file (fname within wolfSSL_SetTmpDH_file_wrapper).

format - an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

Example:

```
Static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
                                      Const char* fname, int format);
long sz = 0;
byte* myBuffer = staticBuffer[FILE_BUFFER_SIZE];
…
if(ssl)
ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
```

See Also:
wolfSSL_SetTmpDH_buffer_wrapper
wc_DhParamsLoad
wolfSSL_SetTmpDH
PemToDer
wolfSSL_CTX_SetTmpDH
wolfSSL_CTX_SetTmpDH_file

wolfSSL_CTX_SetMinRsaKey_Sz

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetMinRsaKey_Sz(WOLFSSL_CTX* ctx, short keySz);

Description:
Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Return Values:
SSL_SUCCESS - returned on successful execution of the function.

BAD_FUNC_ARG - returned if the ctx structure is NULL or the keySz is less than zero or not divisible by 8.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

keySz - a short integer type stored in minRsaKeySz in the **ctx** structure and the **cm** structure converted to bytes.

Example:

```
WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
…
minDhKeyBits = atoi(myoptarg);
…
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
…
```

See Also:
wolfSSL_SetMinRsaKey_Sz

wolfSSL_CTX_SetTmpDH_file

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetTmpDH(WOLFSSL_CTX* ctx, const char* fname, int format);

Description:
The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.

Return Values:
SSL_SUCCESS - returned if the wolfSSL_SetTmpDH_file_wrapper or any of its subroutines return successfully.

MEMORY_E - returned if an allocation of dynamic memory fails in a subroutine.

BAD_FUNC_ARG - returned if the **ctx** or **fname** parameters are NULL or if a subroutine is passed a NULL argument.

SSL_BAD_FILE - returned if the certificate file is unable to open or if the a set of checks on the file fail from wolfSSL_SetTmpDH_file_wrapper.

SSL_BAD_FILETYPE - returned if the format is not PEM or ASN.1 from wolfSSL_SetTmpDH_buffer_wrapper().

DH_KEY_SIZE_E - returned from wolfSSL_SetTmpDH() if the ctx minDhKeySz member exceeds maximum size allowed for DH.

SIDE_ERROR - returned in wolfSSL_SetTmpDH() if the side is not the server end.

SSL_NO_PEM_HEADER - returned from PemToDer if there is no PEM header.

SSL_FATAL_ERROR - returned from PemToDer if there is a memory copy failure.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

fname - a constant character pointer to a certificate file.

format - an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

Example:

```
#define dhParam     “certs/dh2048.pem”
#DEFINE aSSERTiNTne(x, y)     AssertInt(x, y, !=, ==)
WOLFSSL_CTX* ctx;
…
AssertNotNull(ctx = wolfSSL_CTX_new(wolfSSLv23_client_method()))
…
AssertIntNE(SSL_SUCCESS, wolfSSL_CTX_SetTmpDH_file(NULL, dhParam,
                                          SSL_FILETYPE_PEM));
```

See Also:
wolfSSL_SetTmpDH_buffer_wrapper
wolfSSL_SetTmpDH
wolfSSL_CTX_SetTmpDH
wolfSSL_SetTmpDH_buffer
wolfSSL_CTX_SetTmpDH_buffer
wolfSSL_SetTmpDH_file_wrapper
AllocDer
PemToDer

wolfSSL_get_psk_identity_hint

Synopsis:
#include <wolfssl/ssl.h>

const char* wolfSSL_get_psk_identity_hint(const WOLFSSL* ssl);

Description:
This function returns the **psk identity hint**.

Return Values:
const char pointer - the value that was stored in the **arrays** member of the WOLFSSL structure is returned.

NULL - returned if the WOLFSSL or Arrays structures are NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
if(idHint){
	/*The hint was retrieved*/
	return idHint;
} else {
	/*Hint wasn't successfully retrieved */
}
```

See Also:
wolfSSL_get_psk_identity

wolfSSL_SetMinRsaKey_Sz

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetMinRsaKey_Sz(WOLFSSL* ssl, short keySz);

Description:
Sets the minimum allowable key size in bytes for RSA located in the WOLFSSL structure.

Return Values:
SSL_SUCCESS - the minimum was set successfully.

BAD_FUNC_ARG - returned if the ssl structure is NULL or if the ksySz is less than zero or not divisible by 8.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

keySz - a short integer value representing the the minimum key in bits.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
      /*Failed to set. */
}
```

See Also:
wolfSSL_CTX_SetMinRsaKey_Sz

wolfSSL_SetMinDhKey_Sz

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_SetMinDhKey_Sz(WOLFSSL* ssl, word16 keySz);

Description:
Sets the minimum size for a Diffie-Hellman key in the WOLFSSL structure in bytes.

Return Values:
SSL_SUCCESS - the minimum size was successfully set.

BAD_FUNC_ARG - the WOLFSSL structure was NULL or the **keySz** parameter was greater than the allowable size or not divisible by 8.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

keySz - a word16 type representing the bit size of the minimum DH key.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMinDhKey(ssl, keySz) != SSL_SUCCESS){
      /*Failed to set. */
}
```

See Also:
wolfSSL_GetDhKey_Sz

wolfSSL_CTX_set_tmp_dh

Synopsis:
#include <wolfssl/ssl.h>

long wolfSSL_CTX_set_tmp_dh(WOLFSSL_CTX* ctx, WOLFSSL_DH* dh);

Description:
Initializes the WOLFSSL_CTX structure's **dh** member with the Diffie-Hellman

parameters.

Return Values:
SSL_SUCCESS - returned if the function executed successfully.

BAD_FUNC_ARG - returned if the ctx or dh structures are NULL.

SSL_FATAL_ERROR - returned if there was an error setting a structure value.

MEMORY_E - returned if their was a failure to allocate memory.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

dh - a pointer to a WOLFSSL_DH structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL_DH* dh;
…
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

See Also:
wolfSSL_BN_bn2bin

wolfSSL_CTX_use_psk_identity_hint

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_use_psk_identity_hint(WOLFSSL_CTX* ctx, const char* hint);

Description:
This function stores the hint argument in the **server_hint** member of the WOLFSSL_CTX structure.

Return Values:
SSL_SUCCESS - returned for successful execution of the function.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

hint - a constant char pointer that will be copied to the WOLFSSL_CTX structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
const char* hint;
int ret;
…
ret = wolfSSL_CTX_use_psk_identity_hint(ctx, hint);
if(ret == SSL_SUCCESS){
	/* Function was successful. */
	return ret;
} else {
	/*Failure case. */
}
```

See Also:
wolfSSL_use_psk_identity_hint

wolfSSL_use_psk_identity_hint

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_use_psk_identity_hint(WOLFSSL* ssl, const char* hint);

Description:
This function stores the hint argument in the **server_hint** member of the Arrays structure within the WOLFSSL structure.

Return Values:
SSL_SUCCESS - returned if the hint was successfully stored in the WOLFSSL structure.

SSL_FAILURE - returned if the WOLFSSL or Arrays structures are NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

hint - a constant character pointer that holds the hint to be saved in memory.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint; /*pass in valid hint*/
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
      /*Handle failure case. */
}
```

See Also:
wolfSSL_CTX_use_psk_identity_hint

wolfSSL_make_eap_keys

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_make_eap_keys(WOLFSSL* ssl, void* msk, unsigned int len,
const char* label);

Description:
This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.

Return Values:
BUFFER_E - returned if the actual size of the buffer exceeds the maximum size allowable.

MEMORY_E - returned if there is an error with memory allocation.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

msk - a void pointer variable that will hold the result of the p_hash function.

len - an unsigned integer that represents the length of the **msk** variable.

label - a constant char pointer that is copied from in PRF() .

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* msk;
unsigned int len;
const char* label;
…
return wolfSSL_make_eap_keys(ssl, msk, len, label);
```

See Also:
PRF
doPRF
p_hash
wc_HmacFinal
wc_HmacUpdate

wolfSSL_CTX_SetMinEccKey_Sz

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetMinEccKey_Sz(WOLFSSL_CTX* ctx, short keySz);

Description:
Sets the minimum size in bytes for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Return Values:
SSL_SUCCESS - returned for a successful execution and the minEccKeySz member is set.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX struct is NULL or if the **keySz** is negative or not divisible by 8.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

keySz - a short integer type that represents the minimum ECC key size in bits.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
short keySz; /*minimum key size*/
…
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
      /*Failed to set min key size */
}
```

See Also:
wolfSSL_SetMinEccKey_Sz

wolfSSL_SetTmpDH_file

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetTmpDH_file(WOLFSSL* ssl, const char* fname, int format);

Description:
This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

Return Values:
SSL_SUCCESS - returned on successful completion of this function and its subroutines.

MEMORY_E - returned if a memory allocation failed in this function or a subroutine.

SIDE_ERROR - if the **side** member of the **Options** structure found in the WOLFSSL struct is not the server side.

SSL_BAD_FILETYPE - returns if the certificate fails a set of checks.

BAD_FUNC_ARG - returns if an argument value is NULL that is not permitted such as, the WOLFSSL structure.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

fname - a constant char pointer holding the certificate.

format - an integer type that holds the format of the certification.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
…
AssertIntNE(SSL_SUCCESS, wolfSSL_SetTmpDH_file(ssl, dhParam,
SSL_FILETYPE_PEM));
```

See Also:
wolfSSL_CTX_SetTmpDH_file
wolfSSL_SetTmpDH_file_wrapper
wolfSSL_SetTmpDH_buffer
wolfSSL_CTX_SetTmpDH_buffer
wolfSSL_SetTmpDH_buffer_wrapper
wolfSSL_SetTmpDH
wolfSSL_CTX_SetTmpDH

wolfSSL_PubKeyPemToDer

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_PubKeyPemToDer(const unsigned char* pem, int pemSz,
unsigned char* buff, int buffSz);

Description:
Converts the PEM format to DER format.

Return Values:
An **int** type representing the bytes written to buffer.

< 0 - returned for an error.

BAD_FUNC_ARG - returned if the DER length is incorrect or if the pem buff, or buffSz arguments are NULL.

Parameters:

pem - the PEM certificate.

pemSz - the size of the PEM certificate.

buff - the buffer that will be written to from the DerBuffer.

buffSz - the size of the buffer.

Example:

```
unsigned char* pem = "/*pem file*/";
int pemSz = sizeof(pem)/sizeof(char);
unsigned char* buff; /*The buffer*/
int buffSz; /*Initialize*/
...
if(wolfSSL_PubKeyPemToDer(pem, pemSz, buff, buffSz)!= SSL_SUCCESS){
      /*Conversion was not successful */
}
```

See Also:
wolfSSL_PubKeyPemToDer
wolfSSL_PemPubKeyToDer
PemToDer

wolfSSL_CTX_SetTmpDH

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetTmpDH(WOLFSSL_CTX* ctx, const unsigned char* p, int pSz,
Const unsigned char* g, int gSz);

Description:
Sets the parameters for the server CTX Diffie-Hellman.

Return Values:
SSL_SUCCESS - returned if the function and all subroutines return without error.

BAD_FUNC_ARG - returned if the CTX, p or g parameters are NULL.

DH_KEY_SIZE_E - returned if the minDhKeySz member of the WOLFSSL_CTX struct is not the correct size.

MEMORY_E - returned if the allocation of memory failed in this function or a subroutine.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

p - a constant unsigned char pointer loaded into the **buffer** member of the serverDH_P struct.

pSz - an int type representing the size of p, initialized to MAX_DH_SIZE.

g - a constant unsigned char pointer loaded into the **buffer** member of the serverDH_G struct.

gSz - an int type representing the size of g, initialized ot MAX_DH_SIZE.

Example:

```
WOLFSSL_CTX* ctx =  WOLFSSL_CTX_new(/*protocol def*/);
byte* p; /*Initialize / Allocate size*/
byte* g; /*Initialize / Allocate size*/
word32 pSz = (word32)sizeof(p)/sizeof(byte);
word32 gSz = (word32)sizeof(g)/sizeof(byte);
…
int ret =  wolfSSL_CTX_SetTmpDH(ctx, p, pSz, g, gSz);
```

```
if(ret != SSL_SUCCESS){
      /*Failure case*/
}
```

See Also:
wolfSSL_SetTmpDH
wc_DhParamsLoad

wolfSSL_d2i_X509_bio

Synopsis:
#include <wolfssl/ssl.h>

d2i_X509_bio ->

WOLFSSL_X509* wolfSSL_d2i_X509_bio(WOLFSSL_BIO* bio, WOLFSSL_X509** x509);

Description:
This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.

Return Values:
Returns NULL on failure and a new WOLFSSL_X509 structure pointer on success.

Parameters:
bio - pointer to the WOLFSSL_BIO structure that has the DER certificate buffer.

x509 - pointer that get set to new WOLFSSL_X509 structure created.

Example:
```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// load DER into bio
x509 = wolfSSL_d2i_X509_bio(bio, NULL);
Or
wolfSSL_d2i_X509_bio(bio, &x509);
```

```
// use x509 returned (check for NULL)
```

See Also:

wolfSSL_PEM_read_bio_DSAparams

Synopsis:

#include <wolfssl/ssl.h>

PEM_read_bio_DSAparams ->
WOLFSSL_DSA* wolfSSL_PEM_read_bio_DSAparams(WOLFSSL_BIO* bio, WOLFSSL_DSA** x, pem_password_cb* cb, void* u);

Description:

This function get the DSA parameters from a PEM buffer in bio.

Return Values:

On successfully parsing the PEM buffer a WOLFSSL_DSA structure is created and returned. If failing to parse the PEM buffer NULL is returned.

Parameters:

bio - pointer to the WOLFSSL_BIO structure for getting PEM memory pointer.

x - pointer to be set to new WOLFSSL_DSA structure.

cb - password callback function.

u - null terminated password string.

Example:

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// setup bio
dsa = wolfSSL_PEM_read_bio_DSAparams(bio, NULL, NULL, NULL);

// check dsa is not NULL and then use dsa
```

See Also:

wolfSSL_PEM_read_bio_X509_AUX

Synopsis:

#include <wolfssl/ssl.h>

PEM_read_bio_X509_AUX->
WOLFSSL_X509* wolfSSL_PEM_read_bio_X509_AUX(WOLFSSL_BIO* bp,
WOLFSSL_X509** x, pem_password_cb* cb, void* u);

Description:

This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.

Return Values:

On successfully parsing the PEM buffer a WOLFSSL_X509 structure is returned. If unsuccessful NULL is returned.

Parameters:

bp - WOLFSSL_BIO structure to get PEM buffer from.

x - if setting WOLFSSL_X509 by function side effect.

cb - password callback.

u - NULL terminated user password.

Example:

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// setup bio
```

```
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);

//check x509 is not null and then use it
```

See Also:

wolfSSL_PEM_read_bio_X509

wolfSSL_PEM_write_bio_PrivateKey

Synopsis:

#include <wolfssl/ssl.h>

PEM_write_bio_PrivateKey ->
int wolfSSL_PEM_write_bio_PrivateKey(WOLFSSL_BIO* bio, WOLFSSL_EVP_PKEY* key, const WOLFSSL_EVP_CIPHER* cipher, unsigned char* passwd, int len, pem_password_cb* cb, void* arg);

Description:

This function writes a key into a WOLFSSL_BIO structure in PEM format.

Return Values:

On successfully creating the PEM buffer SSL_SUCCESS is returned. If unsuccessful SSL_FAILURE is returned.

Parameters:

bio - WOLFSSL_BIO structure to get PEM buffer from.

key - key to convert to PEM format.

cipher- EVP cipher structure.

passwd - password.

len - length of password.

cb - password callback.

arg - optional argument.

Example:

```
WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// create bio and setup key
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);

//check ret value
```

See Also:

wolfSSL_PEM_read_bio_X509_AUX

wolfSSL_X509_digest

Synopsis:

#include <wolfssl/ssl.h>

X509_digest ->
int wolfSSL_X509_digest(const WOLFSSL_X509* x509, const WOLFSSL_EVP_MD* digest, unsigned char* buf, unsigned int* len)

Description:

This function returns the hash of the DER certificate.

Return Values:

SSL_SUCCESS: On successfully creating a hash.

SSL_FAILURE: Returned on bad input or unsuccessful hash.

Parameters:

x509 - certificate to get the hash of.

digest - the hash algorithm to use.

buf - buffer to hold hash.

len - length of buffer.

Example:

```
WOLFSSL_X509* x509;
unsigned char buffer[64];
unsigned int bufferSz;
int ret;

ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);
//check ret value
```

See Also:

wolfSSL_X509_get_ext_d2i

Synopsis:

#include <wolfssl/ssl.h>

X509_get_ext_d2i ->
void*wolfSSL_X509_get_ext_d2i(const WOLFSSL_X509* x509, int nid, int* c, int* idx)

Description:

This function looks for and returns the extension matching the passed in NID value.

Return Values:

NULL: If extension is not found or error is encountered.

If successful a STACK_OF(WOLFSSL_ASN1_OBJECT) pointer is returned.

Parameters:

x509 - certificate to get parse through for extension.

nid - extension OID to be found.

c - if not NULL is set to -2 for multiple extensions found -1 if not found, 0 if found and not critical and 1 if found and critical.

idx - if NULL return first extension matched otherwise if not stored in x509 start at idx.

Example:

```
const WOLFSSL_X509* x509;
int c;
int idx = 0;
STACK_OF(WOLFSSL_ASN1_OBJECT)* sk;

sk = wolfSSL_X509_get_ext_d2i(x509, NID_basic_constraints, &c, &idx);

//check sk for NULL and then use it. sk needs freed after done.
```

See Also:

wolfSSL_sk_ASN1_OBJECT_free

wolfSSL_X509_NAME_get_text_by_NID

Synopsis:

#include <wolfssl/ssl.h>

X509_NAME_get_text_by_NID ->
int wolfSSL_X509_NAME_get_text_by_NID(WOLFSSL_X509_NAME* name, int nid, char* buf, int len);

Description:

This function gets the text related to the passed in NID value.

Return Values:

Returns the size of text buffer.

Parameters:

name - WOLFSSL_X509_NAME to search for text.

nid - NID to search for.

buf - buffer to hold text when found.

len - length of buffer.

Example:

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// get WOLFSSL_X509_NAME
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName, buffer,
bufferSz);

//check ret value
```

See Also:

wolfSSL_X509_STORE_add_cert

Synopsis:

#include <wolfssl/ssl.h>

X509_STORE_add_cert ->
int wolfSSL_X509_STORE_add_cert(WOLFSSL_X509_STORE* str, WOLFSSL_X509* x509);

Description:

This function adds a certificate to the WOLFSSL_X509_STRE structure.

Return Values:

SSL_SUCCESS: If certificate is added successfully.

SSL_FATAL_ERROR: If certificate is not added successfully.

Parameters:

str - certificate store to add the certificate to.

x509 - certificate to add.

Example:

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;

ret = wolfSSL_X509_STORE_add_cert(str, x509);

//check ret value
```

See Also:

wolfSSL_X509_free

wolfSSL_X509_STORE_CTX_get_chain

Synopsis:

#include <wolfssl/ssl.h>

X509_STORE_CTX_get_chain ->
int wolfSSL_X509_STORE_CTX_get_chain(WOLFSSL_X509_STORE_CTX* ctx);

Description:

This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.

Return Values:

If successful returns WOLFSSL_STACK (same as STACK_OF(WOLFSSL_X509)) pointer otherwise NULL.

Parameters:

ctx - certificate store ctx to get parse chain from.

Example:

```
WOLFSSL_STACK* sk;
WOLFSSL_X509_STORE_CTX* ctx;
```

```
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);

//check sk for NULL and then use it. sk needs freed after done.
```

See Also:
wolfSSL_sk_X509_free

wolfSSL_X509_STORE_set_flags

Synopsis:
#include <wolfssl/ssl.h>

X509_STORE_set_flags ->
int wolfSSL_X509_STORE_set_flags(WOLFSSL_X509_STORE* str, unsigned long flag);

Description:
This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK.

Return Values:
SSL_SUCCESS: If no errors were encountered when setting the flag.

If unsuccessful a negative error value is returned.

Parameters:
str - certificate store to set flag in.

flag - flag for behavior.

Example:
```
WOLFSSL_X509_STORE* str;
int ret;
// create and set up str

ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);
```

```
If (ret != SSL_SUCCESS) {
      //check ret value and handle error case
}
```

See Also:

wolfSSL_X509_STORE_new, wolfSSL_X509_STORE_free

wolfSSL_X509_STORE_CTX_set_flags

Synopsis:

#include <wolfssl/ssl.h>

X509_STORE_CTX_set_flags ->

void wolfSSL_X509_STORE_CTX_set_flags(WOLFSSL_X509_STORE_CTX* ctx, unsigned long flags)

Description:

This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE_CTX. structure passed in.

Return Values:

None

Parameters:

ctx - certificate store CTX to set flag in.

flag - flag for behavior.

Example:

```
WOLFSSL_X509_STORE_CTX* ctx;
// create and set up ctx and flag
wolfSSL_X509_STORE_CTX, set_flags(ctx, flag);
```

See Also:

wolfSSL_X509_STORE_CTX_new, wolfSSL_X509_STORE_CTX_free

wolfSSL_DES_set_key

Synopsis:

<wolfssl/openssl/des.h> is included with:

#include <wolfssl/openssl/des.h>

DES_set_key ->

int wolfSSL_DES_set_key(WOLFSSL_const_DES_cblock* myDes,

WOLFSSL_DES_key_schedule* key);

Description:

This function sets the key schedule. If the macro WOLFSSL_CHECK_DESKEY is defined then acts like wolfSSL_DES_set_key_checked if not then acts like wolfSSL_DES_set_key_unchecked.

Return Values:

If WOLFSSL_CHECK_DESKEY set then -1 if parity error, -2 for weak/null key, and 0 for success.

If macro WOLFSSL_CHECK_DESKEY is not defined then always returns 0.

Parameters:

myDes - DES key

key - key to set from myDes.

Example:

```
WOLFSSL_const_DES_cblock* myDes;
WOLFSSL_DES_key_schedule* key;
int ret;
// load DES key
ret = wolfSSL_DES_set_key(myDes, key);

// check ret value
```

See Also:

wolfSSL_DES_set_key_checked, wolfSSL_DES_set_key_unchecked

wolfSSL_DSA_dup_DH

Synopsis:

#include <wolfssl/ssl.h>

DSA_dup_DH ->

WOLFSSL_DH* wolfSSL_DSA_dup_DH(const WOLFSSL_DSA* dsa);

Description:

This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.

Return Values:

If duplicated returns WOLFSSL_DH structure if function failed NULL is returned.

Parameters:

dsa - WOLFSSL_DSA structure to duplicate.

Example:

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

See Also:

17.3 Context and Session Setup

The functions in this section have to do with creating and setting up SSL/TLS context objects (WOLFSSL_CTX) and SSL/TLS session objects (WOLFSSL).

wolfSSLv3_client_method

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_METHOD *wolfSSLv3_client_method(void);

Description:

The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:

If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Parameters:

This function has no parameters.

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
if (method == NULL) {
      /*unable to get method*/
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfTLSv1_client_method
wolfTLSv1_1_client_method
wolfTLSv1_2_client_method
wolfDTLSv1_client_method
wolfSSLv23_client_method
wolfSSL_CTX_new

wolfSSLv3_server_method

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_METHOD *wolfSSLv3_server_method(void);

Description:
The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Parameters:

This function has no parameters.

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfTLSv1_server_method
wolfTLSv1_1_server_method
wolfTLSv1_2_server_method
wolfDTLSv1_server_method
wolfSSLv23_server_method
wolfSSL_CTX_new

wolfSSLv23_client_method

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_METHOD *wolfSSLv23_client_method(void);

Description:
The wolfSSLv23_client_method() function is used to indicate that the application is a client and will support the highest protocol version supported by the server between SSL 3.0 - TLS 1.2. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well.

To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.2.

Return Values:

If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Parameters:

This function has no parameters.

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_client_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:

wolfSSLv3_client_method
wolfTLSv1_client_method
wolfTLSv1_1_client_method
wolfTLSv1_2_client_method
wolfDTLSv1_client_method
wolfSSL_CTX_new

wolfSSLv23_server_method

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_METHOD *wolfSSLv23_server_method(void);

Description:
The wolfSSLv23_server_method() function is used to indicate that the application is a server and will support clients connecting with protocol version from SSL 3.0 - TLS 1.2. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Parameters:

This function has no parameters.

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_server_method
wolfTLSv1_server_method
wolfTLSv1_1_server_method

wolfTLSv1_2_server_method
wolfDTLSv1_server_method
wolfSSL_CTX_new

wolfTLSv1_client_method

Synopsis:
WOLFSSL_METHOD *wolfTLSv1_client_method(void);

Description:
The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_client_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_client_method
wolfTLSv1_1_client_method
wolfTLSv1_2_client_method
wolfDTLSv1_client_method

wolfSSLv23_client_method
wolfSSL_CTX_new

wolfTLSv1_server_method

Synopsis:
WOLFSSL_METHOD *wolfTLSv1_server_method(void);

Description:
The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
      /*nable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_server_method
wolfTLSv1_1_server_method
wolfTLSv1_2_server_method
wolfDTLSv1_server_method

wolfSSLv23_server_method
wolfSSL_CTX_new

wolfTLSv1_1_client_method

Synopsis:
WOLFSSL_METHOD *wolfTLSv1_1_client_method(void);

Description:
The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_client_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_client_method
wolfTLSv1_client_method
wolfTLSv1_2_client_method
wolfDTLSv1_client_method
wolfSSLv23_client_method

wolfSSL_CTX_new

wolfTLSv1_1_server_method

Synopsis:
WOLFSSL_METHOD *wolfTLSv1_1_server_method(void);

Description:
The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_server_method
wolfTLSv1_server_method
wolfTLSv1_2_server_method
wolfDTLSv1_server_method
wolfSSLv23_server_method
wolfSSL_CTX_new

wolfTLSv1_2_client_method

Synopsis:
WOLFSSL_METHOD *wolfTLSv1_2_client_method(void);

Description:
The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_client_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_client_method
wolfTLSv1_client_method
wolfTLSv1_1_client_method
wolfDTLSv1_client_method
wolfSSLv23_client_method
wolfSSL_CTX_new

wolfTLSv1_2_server_method

Synopsis:
WOLFSSL_METHOD *wolfTLSv1_2_server_method(void);

Description:
The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_server_method
wolfTLSv1_server_method
wolfTLSv1_1_server_method
wolfDTLSv1_server_method
wolfSSLv23_server_method
wolfSSL_CTX_new

wolfSSLv3_server_method_ex; wolfSSLv3_client_method_ex;
wolfTLSv1_server_method_ex; wolfTLSv1_client_method_ex;
wolfTLSv1_1_server_method_ex; wolfTLSv1_1_client_method_ex;
wolfTLSv1_2_server_method_ex; wolfTLSv1_2_client_method_ex;
wolfSSLv23_server_method_ex; wolfSSLv23_client_method_ex;

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap)

Description:

These functions have the same behavior as their counterparts (functions with not having _ex) except that they do not create WOLFSSL_METHOD* using dynamic memory. The functions will use the heap hint passed in to create a new WOLFSSL_METHOD struct.

Return Values:

A value of WOLFSSL_METHOD pointer if success.

NULL is returned in error cases.

Parameters:

heap - a pointer to a heap hint for creating WOLFSSL_METHOD struct.

Example:

```
WOLFSSL_CTX* ctx;
int ret;

...

ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
memory, memorySz, 0, MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
```

```
    // handle error case
}

...
```

See Also:

wolfSSL_new

wolfSSL_CTX_new

wolfSSL_CTX_free

wolfDTLSv1_client_method

Synopsis:

WOLFSSL_METHOD *wolfDTLSv1_client_method(void);

Description:

The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:

If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_client_method
wolfTLSv1_client_method
wolfTLSv1_1_client_method
wolfTLSv1_2_client_method
wolfSSLv23_client_method
wolfSSL_CTX_new

wolfDTLSv1_server_method

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_METHOD *wolfDTLSv1_server_method(void);

Description:
The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Return Values:
If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example:

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
if (method == NULL) {
      /*unable to get method*/
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

See Also:
wolfSSLv3_server_method
wolfTLSv1_server_method
wolfTLSv1_1_server_method
wolfTLSv1_2_server_method
wolfSSLv23_server_method
wolfSSL_CTX_new

wolfSSL_new

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL* wolfSSL_new(WOLFSSL_CTX* ctx);

Description:
This function creates a new SSL session, taking an already created SSL context as input.

Return Values:
If successful the call will return a pointer to the newly-created WOLFSSL structure. Upon failure, NULL will be returned.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

Example:

```
WOLFSSL*     ssl = NULL;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
      /*context creation failed*/
}

ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
```

```
	/*SSL object creation failed*/
}
```

See Also:
wolfSSL_CTX_new

wolfSSL_free

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_free(WOLFSSL* ssl);

Description:
This function frees an allocated WOLFSSL object.

Return Values:
No return values are used for this function.

Parameters:

ssl - pointer to the SSL object, created with wolfSSL_new().

Example:

```
WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

See Also:
wolfSSL_CTX_new
wolfSSL_new
wolfSSL_CTX_free

wolfSSL_ASN1_INTEGER_to_BN

Synopsis:
#include <wolfssl/ssl.h>

ASN1_INTEGER_to_BN ->

WOLFSSL_BIGNUM* wolfSSL_ASN1_INTEGER_to_BN(const WOLFSSL_ASN1_INTEGER* ai, WOLFSSL_BIGNUM* bn);

Description:

This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.

Return Values:

On successfully copying the WOLFSSL_ASN1_INTEGER value a WOLFSSL_BIGNUM pointer is returned.

If a failure occured NULL is returned.

Parameters:

ai - WOLFSSL_ASN1_INTEGER structure to copy from.

bn - if wanting to copy into an already existing WOLFSSL_BIGNUM struct then pass in a pointer to it. Optionally this can be NULL and a new WOLFSSL_BIGNUM structure will be created.

Example:

```
WOLFSSL_ASN1_INTEGER* ai;

WOLFSSL_BIGNUM* bn;

// create ai

bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);

// or if having already created bn and wanting to reuse structure
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);

// check bn is or return value is not NULL
```

See Also:

wolfSSL_ASN1_INTEGER_get

Synopsis:

#include <wolfssl/ssl.h>

#include <l/wolfssl/openssl/asn1.h>

ASN1_INTEGER_get ->
long wolfSSL_ASN1_INTEGER_get(const WOLFSSL_ASN1_INTEGER* i)

Description:

This functions convert ASN1_INTEGER structure to the value.

Return Values:

ASN1_INTEGER_get() returns the value of i but it returns 0 if a is NULL and -1 on error.

Parameters:

i - WOLFSSL_ASN1_INTEGER structure

Example:

```
WOLFSSL_ASN1_INTEGER* i;

long a;

// create ai

a = wolfSSL_ASN1_INTEGER_get(ai);

// check a is or return value is not NULL
```

wolfSSL_BN_mod_exp

Synopsis:

#include <wolfssl/openssl/bn.h>

BN_mod_exp ->
int wolfSSL_BN_mod_exp(WOLFSSL_BIGNUM* r, const WOLFSSL_BIGNUM* a,
const WOLFSSL_BIGNUM* p, const WOLFSSL_BIGNUM* m, WOLFSSL_BN_CTX*
ctx);

Description:

This function performs the following math “r = (a^p) % m”.

Return Values:

SSL_SUCCESS: On successfully performing math operation.

SSL_FAILURE: If an error case was encountered.

Parameters:

r - structure to hold result.

a - value to be raised by a power.

p - power to raise a by.

m - modulus to use.

ctx -currently not used with wolfSSL can be NULL.

Example:

```
WOLFSSL_BIGNUM r,a,p,m;

int ret;

// set big number values

ret  = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// check ret value
```

See Also:

wolfSSL_BN_new, wolfSSL_BN_free

wolfSSL_BN_mod_mul

Synopsis:

#include <wolfssl/openssl/bn.h>

BN_mod_mul ->

nt wolfSSL_BN_mod_mul(WOLFSSL_BIGNUM *r, const WOLFSSL_BIGNUM *a,
    const WOLFSSL_BIGNUM *b, const WOLFSSL_BIGNUM *m,
WOLFSSL_BN_CTX *ctx)

Description:

This function performs the following math “"r=(a*b) mod m”.

Return Values:

SSL_SUCCESS: On successfully performing math operation.

SSL_FAILURE: If an error case was encountered.

Parameters:

r - structure to hold result.

a - value to be multiplied

b - value to multiply

m - modulus to use

ctx -currently not used with wolfSSL can be NULL.

Example:

```
WOLFSSL_BIGNUM r,a,b, m;
int ret;
// set big number values
ret  = wolfSSL_BN_mod_mul(r, a, b, m, NULL);
// check ret value
```

See Also:

wolfSSL_BN_new, wolfSSL_BN_free

wolfSSL_check_private_key

Synopsis:

#include <wolfssl/ssl.h>

SSL_check_private_key ->
int wolfSSL_check_private_key(const WOLFSSL* ssl);

Description:

This function checks that the private key is a match with the certificate being used.

Return Values:

SSL_SUCCESS: On successfully match.

SSL_FAILURE: If an error case was encountered.

All error cases other than SSL_FAILURE are negative values.

Parameters:

ssl - WOLFSSL structure to check.

Example:

```
WOLFSSL* ssl;

int ret;

// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_get_client_random

Synopsis:

#include <wolfssl/ssl.h>

SSL_get_client_random ->
size_t wolfSSL_get_client_random(const WOLFSSL* ssl, unsigned char* out, size_t outSz);

Description:

This is used to get the random data sent by the client during the handshake.

Return Values:

On successfully getting data returns a value greater than 0. If no random data buffer or an error state returns 0. If outSz passed in is 0 then the maximum buffer size needed is returned.

Parameters:

ssl - WOLFSSL structure to get clients random data buffer from.

out -buffer to hold random data.

outSz -size of out buffer passed in. (if 0 function will return max buffer size needed)

Example:

```
WOLFSSL ssl;

unsigned char* buffer;

size_t bufferSz;

size_t ret;

bufferSz  = wolfSSL_get_client_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_client_random(ssl, buffer, bufferSz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_get_server_random

Synopsis:

#include <wolfssl/ssl.h>

SSL_get_server_random ->

size_t wolfSSL_get_server_random(const WOLFSSL* ssl, unsigned char* out, size_t outSz);

Description:

This is used to get the random data sent by the server during the handshake.

Return Values:

On successfully getting data returns a value greater than 0. If no random data buffer or an error state returns 0. If outSz passed in is 0 then the maximum buffer size needed is returned.

Parameters:

ssl - WOLFSSL structure to get clients random data buffer from.

out -buffer to hold random data.

outSz -size of out buffer passed in. (if 0 function will return max buffer size needed)

Example:

```
WOLFSSL ssl;

unsigned char* buffer;

size_t bufferSz;

size_t ret;

bufferSz  = wolfSSL_get_server_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_server_random(ssl, buffer, bufferSz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_SESSION_get_master_key

Synopsis:

#include <wolfssl/ssl.h>

SSL_SESSION_get_master_key ->
int wolfSSL_SESSION_get_master_key(const WOLFSSL_SESSION* ses, unsigned char* out, int outSz);

Description:

This is used to get the master key after completing a handshake.

Return Values:

On successfully getting data returns a value greater than 0. If no data or an error state is hit then the function returns 0. If the outSz passed in is 0 then the maximum buffer size needed is returned.

Parameters:

ses - WOLFSSL_SESSION structure to get master secret buffer from.

out -buffer to hold data.

outSz -size of out buffer passed in. (if 0 function will return max buffer size needed)

Example:

```
WOLFSSL_SESSION ssl;

unsigned char* buffer;

size_t bufferSz;

size_t ret;

// complete handshake and get session structure

bufferSz  = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_SESSION_get_master_key_length

Synopsis:

#include <wolfssl/ssl.h>

SSL_SESSION_get_master_key_length ->
int wolfSSL_SESSION_get_master_key_length(const WOLFSSL_SESSION* ses);

Description:

This is used to get the master secret key length.

Return Values:

Returns master secret key size.

Parameters:

ses - WOLFSSL_SESSION structure to get master secret buffer from.

Example:

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_get_options

Synopsis:

#include <wolfssl/ssl.h>

SSL_get_options ->
unsigned long wolfSSL_get_options(const WOLFSSL* ssl);

Description:

This function returns the current options mask.

Return Values:

Returns the mask value stored in ssl.

Parameters:

ssl - WOLFSSL structure to get options mask from.

Example:

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

See Also:

wolfSSL_new, wolfSSL_free, wolfSSL_set_options

wolfSSL_set_options

Synopsis:

#include <wolfssl/ssl.h>

SSL_set_options ->
unsigned long wolfSSL_get_options(const WOLFSSL* ssl, unsigned long op);

Description:

This function sets the options mask in the ssl.

Some valid options are:

SSL_OP_ALL

SSL_OP_COOKIE_EXCHANGE

SSL_OP_NO_SSLv2

SSL_OP_NO_SSLv3

SSL_OP_NO_TLSv1

SSL_OP_NO_TLSv1_1

SSL_OP_NO_TLSv1_2

SSL_OP_NO_COMPRESSION

Return Values:

Returns the updated options mask value stored in ssl.

Parameters:

ssl - WOLFSSL structure to set options mask.

Example:

```
WOLFSSL* ssl;

unsigned long mask;

mask = SSL_OP_NO_TLSv1

mask  = wolfSSL_set_options(ssl, mask);
// check mask
```

See Also:

wolfSSL_new, wolfSSL_free, wolfSSL_get_options

wolfSSL_set_msg_callback

Synopsis:

#include <wolfssl/ssl.h>

SSL_set_msg_callback ->
int wolfSSL_set_msg_callback(WOLFSSL *ssl, SSL_Msg_Cb cb);

Description:

This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.

Callback function prototype:

typedef void (*SSL_Msg_Cb)(int write_p, int version, int content_type,
    const void *buf, size_t len, WOLFSSL *ssl, void *arg);

Return Values:

SSL_SUCCESS: On success.

SSL_FAILURE: If an NULL ssl passed in.

Parameters:

ssl - WOLFSSL structure to set callback argument.

Example:

```
static cb(int write_p, int version, int content_type,
    const void *buf, size_t len, WOLFSSL *ssl, void *arg)
{ … }

WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
```

```
// check ret
```

See Also:

wolfSSL_set_msg_callback_arg

wolfSSL_set_msg_callback_arg

Synopsis:

#include <wolfssl/ssl.h>

SSL_set_msg_callback_arg ->
void wolfSSL_set_msg_callback_arg(WOLFSSL *ssl, void *arg);

Description:

This function sets associated callback context value in the ssl. The value is handed over to the callback argument.

Return Values:

None

Parameters:

ssl - WOLFSSL structure to set callback argument.

Example:

```
static cb(int write_p, int version, int content_type,
    const void *buf, size_t len, WOLFSSL *ssl, void *arg)
{ … }

WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
       wolfSSL_set_msg_callback(ssl, arg);
```

See Also:

wolfSSL_set_msg_callback

wolfSSL_get_verify_result

Synopsis:

#include <wolfssl/ssl.h>

SSL_get_verify_result ->
long wolfSSL_get_verify_result(const WOLFSSL* ssl);

Description:

This is used to get the results after trying to verify the peer's certificate.

Return Values:

X509_V_OK: On successful verification.

SSL_FAILURE: If an NULL ssl passed in.

Parameters:

ssl - WOLFSSL structure to get verification results from.

Example:

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_get1_session

Synopsis:

#include <wolfssl/ssl.h>

SSL_get1_session ->
WOLFSSL_SESSION* wolfSSL_get1_session(WOLFSSL* ssl)

Description:

This function returns the WOLFSSL_SESSION from the WOLFSSL structure.

Return Values:

WOLFSSL_SESSION: On success return session pointer.

NULL: on failure returns NULL.

Parameters:

ssl - WOLFSSL structure to get session from.

Example:

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;
// attempt/complete handshake
ses  = wolfSSL_get1_session(ssl);
// check ses information
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_set_tlsext_debug_arg

Synopsis:

#include <wolfssl/ssl.h>

SSL_set_tlsext_debug_arg ->
long wolfSSL_set_tlsext_debug_arg(WOLFSSL* ssl, void* arg);

Description:

This is used to set the debug argument passed around.

Return Values:

SSL_SUCCESS: On successful setting argument.

SSL_FAILURE: If an NULL ssl passed in.

Parameters:

ssl - WOLFSSL structure to set argument in.

arg - argument to use.

Example:

```
WOLFSSL* ssl;

void* args;

int ret;

// create ssl object

ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_set_tmp_dh

Synopsis:

#include <wolfssl/ssl.h>

SSL_set_tmp_dh ->
long wolfSSL_set_tmp_dh(WOLFSSL* ssl, WOLFSSL_DH* dh);

Description:

This function sets the temporary DH to use during the handshake.

Return Values:

SSL_SUCCESS: On successful setting DH.

SSL_FAILURE, MEMORY_E, SSL_FATAL_ERROR, BAD_FUNC_ARG: in error cases

Parameters:

ssl - WOLFSSL structure to set temporary DH.

dh - DH to use.

Example:

```
WOLFSSL* ssl;
WOLFSSL_DH* dh;
int ret;
// create ssl object
ret  = wolfSSL_set_tmp_dh(ssl, dh);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_state

Synopsis:

#include <wolfssl/ssl.h>

SSL_state ->
int wolfSSL_state(WOLFSSL* ssl);

Description:

This is used to get the internal error state of the WOLFSSL structure.

Return Values:

Returns ssl error state or BAD_FUNC_ARG if ssl is NULL.

Parameters:

ssl - WOLFSSL structure to get state from.

Example:

```
WOLFSSL* ssl;
```

```
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_use_certificate

Synopsis:

#include <wolfssl/ssl.h>

SSL_use_certificate ->
int wolfSSL_use_certificate(WOLFSSL* ssl, WOLFSSL_X509* x509);

Description:

This is used to set the certificate for WOLFSSL structure to use during a handshake.

Return Values:

SSL_SUCCESS: On successful setting argument.

SSL_FAILURE: If a NULL argument passed in.

Parameters:

ssl - WOLFSSL structure to set certificate in.

x509 - certificate to use.

Example:

```
WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

See Also:
wolfSSL_new, wolfSSL_free

wolfSSL_use_certificate_ASN1

Synopsis:
#include <wolfssl/ssl.h>

SSL_use_certificate_ASN1 ->
int wolfSSL_use_certificate_ASN1(WOLFSSL* ssl, unsigned char* der, int derSz);

Description:
This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.

Return Values:
SSL_SUCCESS: On successful setting argument.

SSL_FAILURE: If a NULL argument passed in.

Parameters:
ssl - WOLFSSL structure to set certificate in.

der - DER certificate to use.

derSz - size of the DER buffer passed in.

Example:
```
WOLFSSL* ssl;
unsigned char* der;
int derSz;
int ret;
// create ssl object and set DER variables
ret  = wolfSSL_use_certificate_ASN1(ssl, der, derSz);
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSLv23_method

Synopsis:

#include <wolfssl/ssl.h>

SSLv23_method ->

WOLFSSL_METHOD* wolfSSLv23_method(void);

Description:

This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

Return Values:

WOLFSSL_METHOD*: On successful creation returns a WOLFSSL_METHOD pointer.

NULL: NULL if memory allocation error or failure to create method.

Parameters:

None

Example:

```
WOLFSSL* ctx;

ctx  = wolfSSL_CTX_new(wolfSSLv23_method());
// check ret value
```

See Also:

wolfSSL_new, wolfSSL_free

wolfSSL_CTX_new

Synopsis:

WOLFSSL_CTX* wolfSSL_CTX_new(WOLFSSL_METHOD* method);

Description:
This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.

Return Values:
If successful the call will return a pointer to the newly-created WOLFSSL_CTX. Upon failure, NULL will be returned.

Parameters:

method - pointer to the desired WOLFSSL_METHOD to use for the SSL context. This is created using one of the wolfSSLvXX_XXXX_method() functions to specify SSL/TLS/DTLS protocol level.

Example:

```
WOLFSSL_CTX*    ctx    = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
if (method == NULL) {
      /*unable to get method*/
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
      /*context creation failed*/
}
```

See Also:
wolfSSL_new

wolfSSL_CTX_free

Synopsis:
void wolfSSL_CTX_free(WOLFSSL_CTX* ctx);

Description:
This function frees an allocated WOLFSSL_CTX object. This function decrements the

CTX reference count and only frees the context when the reference count has reached 0.

Return Values:
No return values are used for this function.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

Example:

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

See Also:
wolfSSL_CTX_new
wolfSSL_new
wolfSSL_free

wolfSSL_CTX_clear_options

Synopsis:
long wolfSSL_CTX_clear_options(WOLFSSL_CTX* ctx, long opt);

Description:
This function resets option bits of WOLFSSL_CTX object.

Return Values:
New option bits

Parameters:

ctx - pointer to the SSL context.

Example:

```
WOLFSSL_CTX* ctx = 0;
...
```

```
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

See Also:
wolfSSL_CTX_new
wolfSSL_new
wolfSSL_free

wolfSSL_CTX_add_extra_chain_cert

Synopsis:
#include <wolfssl/ssl.h>

SSL_CTX_add_extra_chain_cert ->
long wolfSSL_CTX_add_extra_chain_cert(WOLFSSL_CTX* ctx, WOLFSSL_X509* x509);

Description:
This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.

Return Values:
SSL_SUCCESS: after successfully adding the certificate.

SSL_FAILURE: if failing to add the certificate to the chain.

Parameters:
ctx - WOLFSSL_CTX structure to add certificate to.

x509 - certificate to add to the chain.

Example:
```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
```

```
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free

wolfSSL_CTX_get_cert_store

Synopsis:

#include <wolfssl/ssl.h>

SSL_CTX_get_cert_store ->

WOLFSSL_X509_STORE* wolfSSL_CTX_get_cert_store(WOLFSSL_CTX* ctx);

Description:

This is a getter function for the WOLFSSL_X509_STORE structure in ctx.

Return Values:

WOLFSSL_X509_STORE*: On successfully getting the pointer.

NULL: Returned if NULL arguments are passed in.

Parameters:

ctx - pointer to the WOLFSSL_CTX structure for getting cert store pointer.

Example:

```
WOLFSSL_CTX ctx;

WOLFSSL_X509_STORE* st;

// setup ctx

st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free, wolfSSL_CTX_set_cert_store

wolfSSL_CTX_set_cert_store

Synopsis:

#include <wolfssl/ssl.h>

SSL_CTX_set_cert_store ->
void wolfSSL_CTX_set_cert_store(WOLFSSL_CTX* ctx, WOLFSSL_X509_STORE* str);

Description:

This is a setter function for the WOLFSSL_X509_STORE structure in ctx.

Return Values:

None

Parameters:

ctx - pointer to the WOLFSSL_CTX structure for setting cert store pointer.

str - pointer to the WOLFSSL_X509_STORE to set in ctx.

Example:

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free, wolfSSL_CTX_get_cert_store

wolfSSL_CTX_get_default_passwd_cb

Synopsis:

#include <wolfssl/ssl.h>

SSL_CTX_get_default_passwd_cb ->
int wolfSSL_CTX_get_default_passwd_cb(WOLFSSL_CTX* ctx)

Description:
This is a getter function for the password callback set in ctx.

Return Values:
On success returns the callback function.

NULL: If ctx is NULL then NULL is returned.

Parameters:
ctx - WOLFSSL_CTX structure to get call back from.

Example:
```
WOLFSSL_CTX* ctx;
Pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);

//use cb
```

See Also:
wolfSSL_CTX_new, wolfSSL_CTX_free

wolfSSL_CTX_get_default_passwd_cb_userdata

Synopsis:
#include <wolfssl/ssl.h>

SSL_CTX_get_default_passwd_cb_userdata ->
void* wolfSSL_CTX_get_default_passwd_cb_userdata(WOLFSSL_CTX* ctx)

Description:
This is a getter function for the password callback user data set in ctx.

Return Values:
On success returns the user data pointer.

NULL: If ctx is NULL then NULL is returned.

Parameters:
ctx - WOLFSSL_CTX structure to get user data from.

Example:
```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx

data = wolfSSL_CTX_get_default_passwd_cb(ctx);

//use data
```

See Also:
wolfSSL_CTX_new, wolfSSL_CTX_free

wolfSSL_CTX_get_read_ahead

Synopsis:
#include <wolfssl/ssl.h>

SSL_CTX_get_read_ahead ->
int wolfSSL_CTX_get_read_ahead(WOLFSSL_CTX* ctx);

Description:
This function returns the get read ahead flag from a WOLFSSL_CTX structure;

Return Values:
On success returns the read ahead flag.

SSL_FAILURE: If ctx is NULL then SSL_FAILURE is returned.

Parameters:

ctx - WOLFSSL_CTX structure to get read ahead flag from.

Example:

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);

//check flag
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free, wolfSSL_CTX_set_read_ahead

wolfSSL_CTX_set_read_ahead

Synopsis:

#include <wolfssl/ssl.h>

SSL_CTX_set_read_ahead ->
int wolfSSL_CTX_set_read_ahead(WOLFSSL_CTX* ctx, int v);

Description:

This function sets the read ahead flag in the WOLFSSL_CTX structure;

Return Values:

SSL_SUCCESS: If ctx read ahead flag set.

SSL_FAILURE: If ctx is NULL then SSL_FAILURE is returned.

Parameters:

ctx - WOLFSSL_CTX structure to set read ahead flag.

Example:

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free, wolfSSL_CTX_get_read_ahead

wolfSSL_CTX_set_tlsext_status_arg

Synopsis:

#include <wolfssl/ssl.h>

SSL_CTX_set_tlsext_status_arg ->
long wolfSSL_CTX_set_tlsext_status_arg(WOLFSSL_CTX* ctx, void* arg);

Description:

This function sets the options argument to use with OCSP.

Return Values:

SSL_FAILURE: If ctx or it’s cert manager is NULL.

SSL_SUCCESS: If successfully set.

Parameters:

ctx - WOLFSSL_CTX structure to set user argument.

arg - user argument.

Example:

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
```

```
// setup ctx
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//check ret value
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free

wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg

Synopsis:

#include <wolfssl/ssl.h>

SSL_CTX_set_tlsext_opaque_prf_input_callback_arg ->
long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(WOLFSSL_CTX* ctx, void* arg);

Description:

This function sets the optional argument to be passed to the PRF callback.

Return Values:

SSL_FAILURE: If ctx is NULL.

SSL_SUCCESS: If successfully set.

Parameters:

ctx - WOLFSSL_CTX structure to set user argument.

arg - user argument.

Example:

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
```

```
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);

//check ret value
```

See Also:

wolfSSL_CTX_new, wolfSSL_CTX_free

wolfSSL_SetVersion

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_SetVersion(WOLFSSL* ssl, int version);

Description:

This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by **version**.

This will override the protocol setting for the SSL session (**ssl**) - originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.

Return Values:

If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG will be returned if the input SSL object is NULL or an incorrect protocol version is given for **version**.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

version - SSL/TLS protocol version. Possible values include WOLFSSL_SSLV3, WOLFSSL_TLSV1, WOLFSSL_TLSV1_1, WOLFSSL_TLSV1_2.

Example:

```
int ret = 0;
WOLFSSL* ssl;
...
```

```
ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
      /*failed to set SSL session protocol version*/
}
```

See Also:
wolfSSL_CTX_new

wolfSSL_use_old_poly

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_use_old_poly(WOLFSSL* ssl, int value);

Description:
Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.

Return Values:
If successful the call will return **0**.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

value - whether or not to use the older version of setting up the information for poly1305. Passing a flag value of 1 indicates yes use the old poly AEAD, to switch back to using the new version pass a flag value of 0.

Example:

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
      /*failed to set poly1305 AEAD version*/
}
```

wolfSSL_check_domain_name

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn);

Description:
wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() or wolfSSL_accept() will add a domain name check to the list of checks to perform. **dn** holds the domain name to check against the peer certificate when it's received.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_FAILURE will be returned if a memory error was encountered.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

dn - domain name to check against the peer certificate when received.

Example:

```
int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
      /*failed to enable domain name check*/
}
```

See Also:
NA

wolfSSL_set_cipher_list

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_set_cipher_list(WOLFSSL* ssl, const char* list);

Description:
This function sets cipher suite list for a given WOLFSSL object (SSL session). The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_set_cipher_list() resets the cipher suite list for the specific SSL session to the provided list each time the function is called.

The cipher suite list, **list**, is a null-terminated text string, and a colon-delimited list. For example, one value for **list** may be

"DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256"

Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c):

RC4-SHA
RC4-MD5
DES-CBC3-SHA
AES128-SHA
AES256-SHA
NULL-SHA
NULL-SHA256
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
PSK-AES128-CBC-SHA256
PSK-AES128-CBC-SHA
PSK-AES256-CBC-SHA
PSK-NULL-SHA256
PSK-NULL-SHA
HC128-MD5
HC128-SHA
HC128-B2B256
AES128-B2B256

AES256-B2B256
RABBIT-SHA
NTRU-RC4-SHA
NTRU-DES-CBC3-SHA
NTRU-AES128-SHA
NTRU-AES256-SHA
QSH
AES128-CCM-8
AES256-CCM-8
ECDHE-ECDSA-AES128-CCM-8
ECDHE-ECDSA-AES256-CCM-8
ECDHE-RSA-AES128-SHA
ECDHE-RSA-AES256-SHA
ECDHE-ECDSA-AES128-SHA
ECDHE-ECDSA-AES256-SHA
ECDHE-RSA-RC4-SHA
ECDHE-RSA-DES-CBC3-SHA
ECDHE-ECDSA-RC4-SHA
ECDHE-ECDSA-DES-CBC3-SHA
AES128-SHA256
AES256-SHA256
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
ECDH-RSA-AES128-SHA
ECDH-RSA-AES256-SHA
ECDH-ECDSA-AES128-SHA
ECDH-ECDSA-AES256-SHA
ECDH-RSA-RC4-SHA
ECDH-RSA-DES-CBC3-SHA
ECDH-ECDSA-RC4-SHA
ECDH-ECDSA-DES-CBC3-SHA
AES128-GCM-SHA256
AES256-GCM-SHA384
DHE-RSA-AES128-GCM-SHA256
DHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES256-GCM-SHA384
ECDH-RSA-AES128-GCM-SHA256

ECDH-RSA-AES256-GCM-SHA384
ECDH-ECDSA-AES128-GCM-SHA256
ECDH-ECDSA-AES256-GCM-SHA384
CAMELLIA128-SHA
DHE-RSA-CAMELLIA128-SHA
CAMELLIA256-SHA
DHE-RSA-CAMELLIA256-SHA
CAMELLIA128-SHA256
DHE-RSA-CAMELLIA128-SHA256
CAMELLIA256-SHA256
DHE-RSA-CAMELLIA256-SHA256
ECDHE-RSA-AES128-SHA256
ECDHE-ECDSA-AES128-SHA256
ECDH-RSA-AES128-SHA256
ECDH-ECDSA-AES128-SHA256
ECDHE-ECDSA-AES256-SHA384
ECDH-RSA-AES256-SHA384
ECDH-ECDSA-AES256-SHA384

Return Values:

SSL_SUCCESS will be returned upon successful function completion, otherwise **SSL_FAILURE** will be returned on failure.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

list - null-terminated text string and a colon-delimited list of cipher suites to use with the specified SSL session.

Example:

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_cipher_list(ssl,
"DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256");
if (ret != SSL_SUCCESS) {
      /*failed to set cipher suite list*/
}
```

See Also:
wolfSSL_CTX_set_cipher_list
wolfSSL_new

wolfSSL_CTX_set_cipher_list

Synopsis:
int wolfSSL_CTX_set_cipher_list(WOLFSSL_CTX* ctx, const char* list);

Description:
This function sets cipher suite list for a given WOLFSSL_CTX. This cipher suite list becomes the default list for any new SSL sessions (WOLFSSL) created using this context. The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_CTX_set_cipher_list() resets the cipher suite list for the specific SSL context to the provided list each time the function is called.

The cipher suite list, **list**, is a null-terminated text string, and a colon-delimited list. For example, one value for **list** may be

"DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256"

Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c):

RC4-SHA
RC4-MD5
DES-CBC3-SHA
AES128-SHA
AES256-SHA
NULL-SHA
NULL-SHA256
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
PSK-AES128-CBC-SHA256
PSK-AES128-CBC-SHA
PSK-AES256-CBC-SHA
PSK-NULL-SHA256
PSK-NULL-SHA

HC128-MD5
HC128-SHA
HC128-B2B256
AES128-B2B256
AES256-B2B256
RABBIT-SHA
QSH
AES128-CCM-8
AES256-CCM-8
ECDHE-ECDSA-AES128-CCM-8
ECDHE-ECDSA-AES256-CCM-8
ECDHE-RSA-AES128-SHA
ECDHE-RSA-AES256-SHA
ECDHE-ECDSA-AES128-SHA
ECDHE-ECDSA-AES256-SHA
ECDHE-RSA-RC4-SHA
ECDHE-RSA-DES-CBC3-SHA
ECDHE-ECDSA-RC4-SHA
ECDHE-ECDSA-DES-CBC3-SHA
AES128-SHA256
AES256-SHA256
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
ECDH-RSA-AES128-SHA
ECDH-RSA-AES256-SHA
ECDH-ECDSA-AES128-SHA
ECDH-ECDSA-AES256-SHA
ECDH-RSA-RC4-SHA
ECDH-RSA-DES-CBC3-SHA
ECDH-ECDSA-RC4-SHA
ECDH-ECDSA-DES-CBC3-SHA
AES128-GCM-SHA256
AES256-GCM-SHA384
DHE-RSA-AES128-GCM-SHA256
DHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES256-GCM-SHA384
ECDH-RSA-AES128-GCM-SHA256

ECDH-RSA-AES256-GCM-SHA384
ECDH-ECDSA-AES128-GCM-SHA256
ECDH-ECDSA-AES256-GCM-SHA384
CAMELLIA128-SHA
DHE-RSA-CAMELLIA128-SHA
CAMELLIA256-SHA
DHE-RSA-CAMELLIA256-SHA
CAMELLIA128-SHA256
DHE-RSA-CAMELLIA128-SHA256
CAMELLIA256-SHA256
DHE-RSA-CAMELLIA256-SHA256
ECDHE-RSA-AES128-SHA256
ECDHE-ECDSA-AES128-SHA256
ECDH-RSA-AES128-SHA256
ECDH-ECDSA-AES128-SHA256
ECDHE-ECDSA-AES256-SHA384
ECDH-RSA-AES256-SHA384
ECDH-ECDSA-AES256-SHA384

Return Values:

SSL_SUCCESS will be returned upon successful function completion, otherwise **SSL_FAILURE** will be returned on failure.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

list - null-terminated text string and a colon-delimited list of cipher suites to use with the specified SSL context.

Example:

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_cipher_list(ctx,
“DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256”);
if (ret != SSL_SUCCESS) {
      /*failed to set cipher suite list*/
}
```

See Also:
wolfSSL_set_cipher_list
wolfSSL_CTX_new

Synopsis:
#include <wolfssl/ssl.h>

EVP_aes_128_ecb ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_aes_128_ecb(void);

EVP_aes_192_ecb ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_aes_192_ecb(void);

EVP_aes_256_ecb ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_aes_256_ecb(void);

Description:
Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings.

Return Values:
Returns a WOLFSSL_EVP_CIPHER pointer.

Parameters:
None

Example:
```
WOLFSSL_EVP_CIPHER* cipher;
```

```
cipher = wolfSSL_EVP_aes_192_ecb();
….
```

See Also:

wolfSSL_EVP_CIPHER_CTX_init

wolfSSL_EVP_CIPHER_block_size

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CIPHER_block_size ->
int wolfSSL_EVP_CIPHER_block_size(const WOLFSSL_EVP_CIPHER* cipher);

Description:

This is a getter function for the block size of cipher.

Return Values:

Returns the block size.

Parameters:

cipher - cipher to get block size of.

Example:

```
printf("block size = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

See Also:

wolfSSL_EVP_aes_256_ctr

wolfSSL_EVP_CIPHER_CTX_block_size

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CIPHER_CTX_block_size ->
int wolfSSL_EVP_CIPHER_CTX_block_size(const WOLFSSL_EVP_CIPHER_CTX* ctx);

Description:

This is a getter function for the ctx block size.

Return Values:

Returns ctx->block_size.

Parameters:

ctx - the cipher ctx to get block size of.

Example:

```
const WOLFSSL_CVP_CIPHER_CTX* ctx;
//set up ctx
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

See Also:

wolfSSL_EVP_CIPHER_block_size

wolfSSL_EVP_CIPHER_CTX_set_flags

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CIPHER_CTX_set_flags ->

void wolfSSL_EVP_CIPHER_CTX_set_flags(WOLFSSL_EVP_CIPHER_CTX* ctx, int flags);

Description:

Setter function for WOLFSSL_EVP_CIPHER_CTX structure.

Return Values:

None

Parameters:

ctx - structure to set flag.

flag - flag to set in structure.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx

wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

See Also:

wolfSSL_EVP_CIPHER_flags

wolfSSL_EVP_CIPHER_CTX_set_key_length

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CIPHER_CTX_set_key_length ->
int wolfSSL_EVP_CIPHER_CTX_set_key_length(WOLFSSL_EVP_CIPHER_CTX* ctx, int keylen);

Description:

Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.

Return Values:

SSL_SUCCESS: If successfully set.

SSL_FAILURE: If failed to set key length/

Parameters:

ctx - structure to set key length.

keylen - key length.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

See Also:

wolfSSL_EVP_CIPHER_flags

wolfSSL_EVP_CIPHER_CTX_set_padding

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CIPHER_CTX_set_padding ->
int wolfSSL_EVP_CIPHER_CTX_set_padding(WOLFSSL_EVP_CIPHER_CTX* ctx, int padding);

Description:

Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.

Return Values:

SSL_SUCCESS: If successfully set.

BAD_FUNC_ARG: If null argument passed in.

Parameters:

ctx - structure to set padding flag.

padding - 0 for not setting padding, 1 for setting padding.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

See Also:

wolfSSL_EVP_CIPHER_flags

wolfSSL_EVP_CipherFinal

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CipherFinal ->
int wolfSSL_EVP_CipherFinal(WOLFSSL_EVP_CIPHER_CTX* ctx, unsigned char* out, int* out1);

Description:

This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.

Return Values:

1:Returned on success

0: If encountering a failure.

Parameters:

ctx - structure to decrypt/encrypt with.

out - buffer for final decrypt/encrypt.

out1 - size of out buffer when data has been added by function.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// create ctx

wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

See Also:

wolfSSL_EVP_CIPHER_CTX_new

wolfSSL_CipherInit_ex

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CipherInit_ex ->
int wolfSSL_CipherInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx, const WOLFSSL_EVP_CIPHER* type, WOLFSSL_ENGINE* impl, unsigned char* key, unsigned char* iv, int enc);

Description:

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.

Return Values:

SSL_SUCCESS: If successfully set.

SSL_FAILURE: If not successful.

Parameters:

ctx - structure to initialize.

type - type of encryption/decryption to do, for example AES.

impl - engine to use. N/A for wolfSSL, can be NULL.

key - key to set .

iv - iv if needed by algorithm.

enc - encryption (1) or decryption (0) flag.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];

wolfCrypt_Init();

ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
 printf("issue creating ctx\n");
 return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_CipherInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_CipherInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

See Also:

wolfSSL_EVP_CIPHER_CTX_new, wolfCrypt_Init, wolfSSL_EVP_CIPHER_CTX_free

wolfSSL_EVP_CipherUpdate

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_CipherUpdate ->
int wolfSSL_EVP_CipherUpdate(WOLFSSL_EVP_CIPHER_CTX* ctx, unsigned char* out, int *outl, const unsigned char* in, int inl);

Description:

Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.

Return Values:

SSL_SUCCESS: If successfull.

SSL_FAILURE: If not successful.

Parameters:

ctx - structure to get cipher type from.

out - buffer to hold output.

outl - adjusted to be size of output.

in - buffer to perform operation on.

inl - length of input buffer.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100];
int outl;
unsigned char in[100];
int inl = 100;
```

```
ctx = wolfSSL_EVP_CIPHER_CTX_new();
// set up ctx
ret = wolfSSL_EVP_CipherUpdate(ctx, out, outl, in, inl);
// check ret value
// buffer out holds outl bytes of data
// free resources
```

See Also:

wolfSSL_EVP_CIPHER_CTX_new, wolfCrypt_Init, wolfSSL_EVP_CIPHER_CTX_free

wolfSSL_EVP_DecryptInit_ex

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_DecryptInit_ex ->
int wolfSSL_EVP_DecryptInit_ex

Description:

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.

Return Values:

SSL_SUCCESS: If successfully set.

SSL_FAILURE: If not successful.

Parameters:

ctx - structure to initialize.

type - type of encryption/decryption to do, for example AES.

impl - engine to use. N/A for wolfSSL, can be NULL.

key - key to set .

iv - iv if needed by algorithm.

enc - encryption (1) or decryption (0) flag.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];

wolfCrypt_Init();

ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
 printf("issue creating ctx\n");
 return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_DecryptInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_DecryptInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

See Also:

wolfSSL_EVP_CIPHER_CTX_new, wolfCrypt_Init, wolfSSL_EVP_CIPHER_CTX_free

wolfSSL_EVP_des_cbc, wolfSSL_EVP_des_ecb

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_des_cbc ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_cbc(void);

EVP_des_ecb ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_ecb(void);

Description:

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers.

wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().

Return Values:

Returns a WOLFSSL_EVP_CIPHER pointer for DES operations.

Parameters:

None

Example:

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_ecb();

….
```

See Also:

wolfSSL_EVP_CIPHER_CTX_init

wolfSSL_EVP_des_cbc, wolfSSL_EVP_des_ecb

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_des_ede3_cbc ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_ede_cbc(void);

EVP_des_ede3_ecb ->

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_ede3_ecb(void);

Description:

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().

Return Values:

Returns a WOLFSSL_EVP_CIPHER pointer for DES EDE3 operations.

Parameters:

None

Example:

```
printf("block size des ede3 cbc = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_cbc()));
printf("block size des ede3 ecb = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_ecb()));
```

See Also:

wolfSSL_EVP_CIPHER_CTX_init

wolfSSL_EVP_DigestInit_ex

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_DigestInit_ex ->
int wolfSSL_EVP_DigestInit_ex(WOLFSSL_EVP_MD_CTX* ctx, const WOLFSSL_EVP_MD* type, WOLFSSL_ENGINE* impl);

Description:

Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.

Return Values:

SSL_SUCCESS: If successfully set.

SSL_FAILURE: If not successful.

Parameters:

ctx - structure to initialize.

type - type of hash to do, for example SHA.

impl - engine to use. N/A for wolfSSL, can be NULL.

Example:

```
WOLFSSL_EVP_MD_CTX* md = NULL;

wolfCrypt_Init();

md = wolfSSL_EVP_MD_CTX_new();
if (md == NULL) {
  printf("error setting md\n");
  return -1;
}

printf("cipher md init ret = %d\n", wolfSSL_EVP_DigestInit_ex(md,
wolfSSL_EVP_sha1(), e));

//free resources
```

See Also:

wolfSSL_EVP_MD_CTX_new, wolfCrypt_Init, wolfSSL_EVP_MD_CTX_free

wolfSSL_EVP_EncryptInit_ex

Synopsis:

#include <wolfssl/openssl/evp.h>

EVP_EncryptInit_ex ->
int wolfSSL_EVP_EncryptInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx, const WOLFSSL_EVP_Cipher* type, WOLFSSL_ENGINE* impl, unsigned char* key, unsigned char* iv);

Description:

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.

Return Values:

SSL_SUCCESS: If successfully set.

SSL_FAILURE: If not successful.

Parameters:

ctx - structure to initialize.

type - type of encryption to do, for example AES.

impl - engine to use. N/A for wolfSSL, can be NULL.

key - key to use.

iv - iv to use.

Example:

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;

wolfCrypt_Init();

ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
  printf("error setting ctx\n");
  return -1;
}
```

```
printf("cipher ctx init ret = %d\n", wolfSSL_EVP_EncryptInit_ex(ctx,
wolfSSL_EVP_aes_128_cbc(), e, key, iv));

//free resources
```

See Also:

wolfSSL_EVP_CIPHER_CTX_new, wolfCrypt_Init, wolfSSL_EVP_CIPHER_CTX_free

wolfSSL_set_compression

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_set_compression(WOLFSSL* ssl);

Description:

Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use **--with-libz** for the configure system and define HAVE_LIBZ otherwise.

Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

Return Values:

If successful the call will return **SSL_SUCCESS**.

NOT_COMPILED_IN will be returned if compression support wasn't built into the library.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int ret = 0;
```

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
      /*successfully enabled compression for SSL session*/
}
```

See Also:
NA

wolfSSL_set_fd

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_set_fd(WOLFSSL* ssl, int fd);

Description:
This function assigns a file descriptor (**fd**) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.

Return Values:
If successful the call will return **SSL_SUCCESS**, otherwise, **Bad_FUNC_ARG** will be returned.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

fd - file descriptor to use with SSL/TLS connection.

Example:

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
      /*failed to set SSL file descriptor*/
```

```
}
```

See Also:
wolfSSL_SetIOSend
wolfSSL_SetIORecv
wolfSSL_SetIOReadCtx
wolfSSL_SetIOWriteCtx

wolfSSL_set_group_messages

Synopsis:
int wolfSSL_set_group_messages(WOLFSSL* ssl);

Description:
This function turns on grouping of handshake messages where possible.

Return Values:

SSL_SUCCESS will be returned upon success.

BAD_FUNC_ARG will be returned if the input context is null.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
      // failed to set handshake message grouping
}
```

See Also:
wolfSSL_CTX_set_group_messages
wolfSSL_new

wolfSSL_CTX_set_group_messages

Synopsis:
int wolfSSL_CTX_set_group_messages(WOLFSSL_CTX* ctx);

Description:
This function turns on grouping of handshake messages where possible.

Return Values:

SSL_SUCCESS will be returned upon success.

BAD_FUNC_ARG will be returned if the input context is null.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

Example:

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
      /*failed to set handshake message grouping*/
}
```

See Also:
wolfSSL_set_group_messages
wolfSSL_CTX_new

wolfSSL_set_session

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_set_session(WOLFSSL* ssl, WOLFSSL_SESSION* session);

Description:
This function sets the session to be used when the SSL object, **ssl**, is used to establish a SSL/TLS connection.

For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default.

Return Values:

SSL_SUCCESS will be returned upon successfully setting the session.

SSL_FAILURE will be returned on failure. This could be caused by the session cache being disabled, or if the session has timed out.

Parameters:

ssl - pointer to the SSL object, created with wolfSSL_new().

session - pointer to the WOLFSSL_SESSION used to set the session for **ssl**.

Example:

```
int ret = 0;
WOLFSSL* ssl = 0;
WOLFSSL_SESSION* session;
...

ret = wolfSSL_get_session(ssl, session);
if (ret != SSL_SUCCESS) {
      /*failed to set the SSL session*/
}
...
```

See Also:

wolfSSL_get_session

wolfSSL_CTX_set_session_cache_mode

Synopsis:
long wolfSSL_CTX_set_session_cache_mode(WOLFSSL_CTX* ctx, long mode);

Description:
This function enables or disables SSL session caching. Behavior depends on the value used for **mode**. The following values for **mode** are available:

SSL_SESS_CACHE_OFF
- disable session caching. Session caching is turned on by default.

SSL_SESS_CACHE_NO_AUTO_CLEAR
- Disable auto-flushing of the session cache. Auto-flushing is turned on by default.

Return Values:

SSL_SUCCESS will be returned upon success.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

mode - modifier used to change behavior of the session cache.

Example:

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
if (ret != SSL_SUCCESS) {
      /*failed to turn SSL session caching off*/
}
```

See Also:
wolfSSL_flush_sessions
wolfSSL_get_session

wolfSSL_set_session
wolfSSL_get_sessionID
wolfSSL_CTX_set_timeout

wolfSSL_set_timeout

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_set_timeout(WOLFSSL* ssl, unsigned int to);

Description:
This function sets the SSL session timeout value in seconds.

Return Values:

SSL_SUCCESS will be returned upon successfully setting the session.

BAD_FUNC_ARG will be returned if **ssl** is NULL.

Parameters:

ssl - pointer to the SSL object, created with wolfSSL_new().

to - value, in seconds, used to set the SSL session timeout.

Example:

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
      /*failed to set session timeout value*/
}
...
```

See Also:
wolfSSL_get_session
wolfSSL_set_session

wolfSSL_CTX_set_timeout

Synopsis:
int wolfSSL_CTX_set_timeout(WOLFSSL_CTX* ctx, unsigned int to);

Description:
This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.

Return Values:

SSL_SUCCESS will be returned upon success.

BAD_FUNC_ARG will be returned when the input context (**ctx**) is null.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

to - session timeout value in seconds

Example:

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
      /*failed to set session timeout value*/
}
```

See Also:
wolfSSL_flush_sessions
wolfSSL_get_session
wolfSSL_set_session
wolfSSL_get_sessionID
wolfSSL_CTX_set_session_cache_mode

wolfSSL_set_using_nonblock

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_set_using_nonblock(WOLFSSL* ssl, int nonblock);

Description:

This function informs the WOLFSSL object that the underlying I/O is non-blocking.

After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Return Values:

This function does not have a return value.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

nonblock - value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

Example:

```
WOLFSSL* ssl = 0;
...

wolfSSL_set_using_nonblock(ssl, 1);
```

See Also:

wolfSSL_get_using_nonblock
wolfSSL_dtls_got_timeout
wolfSSL_dtls_get_current_timeout

wolfSSL_set_verify

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_set_verify(WOLFSSL* ssl, int mode, VerifyCallback vc);

typedef int (*VerifyCallback)(int, WOLFSSL_X509_STORE_CTX*);

Description:
This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for **verify_callback**.

The verification **mode** of peer certificates is a logically OR'd list of flags. The possible flag values include:

SSL_VERIFY_NONE

> **Client mode**: the client will not verify the certificate received from the server and the handshake will continue as normal.
>
> **Server mode**: the server will not send a certificate request to the client. As such, client verification will not be enabled.

SSL_VERIFY_PEER

> **Client mode**: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect.
>
> **Server mode**: the server will send a certificate request to the client and verify the client certificate received.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

> **Client mode**: no effect when used on the client side.
>
> **Server mode**: the verification will fail on the server side if the client fails to send

a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server).

SSL_VERIFY_FAIL_EXCEPT_PSK

Client mode: no effect when used on the client side.

Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

Return Values:

This function has no return value.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

mode - session timeout value in seconds

verify_callback - callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

Example:

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT,
0);
```

See Also:
wolfSSL_CTX_set_verify

wolfSSL_CTX_set_verify

Synopsis:
void wolfSSL_CTX_set_verify(WOLFSSL_CTX* ctx, int mode,
VerifyCallback vc);

typedef int (*VerifyCallback)(int, WOLFSSL_X509_STORE_CTX*);

Description:
This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for **verify_callback**.

The verification **mode** of peer certificates is a logically OR'd list of flags. The possible flag values include:

SSL_VERIFY_NONE

> **Client mode**: the client will not verify the certificate received from the server and the handshake will continue as normal.
>
> **Server mode**: the server will not send a certificate request to the client. As such, client verification will not be enabled.

SSL_VERIFY_PEER

> **Client mode**: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect.
>
> **Server mode**: the server will send a certificate request to the client and verify the client certificate received.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

> **Client mode**: no effect when used on the client side.
>
> **Server mode**: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the

SSL server).

SSL_VERIFY_FAIL_EXCEPT_PSK

Client mode: no effect when used on the client side.

Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

Return Values:

This function has no return value.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

mode - session timeout value in seconds

verify_callback - callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

Example:

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, SSL_VERIFY_PEER |
                       SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

See Also:

wolfSSL_set_verify

wolfSSL_CTX_get_verify_depth

Synopsis:

#include <wolfssl/ssl.h>

long wolfSSL_CTX_get_verify_depth(WOLFSSL_CTX* ctx);

Description:
This function gets the certificate chaining depth using the CTX structure.

Return Values:
MAX_CHAIN_DEPTH - returned if the CTX struct is not NULL. The constant representation of the max certificate chain peer depth.

BAD_FUNC_ARG - returned if the CTX structure is NULL.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL_METHOD method; /*protocol method*/
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
	/*You have the expected value*/
} else {
	/*Handle an unexpected depth*/
}
```

See Also:
wolfSSL_CTX_use_certificate_chain_file
wolfSSL_get_verify_depth

wolfSSL_CTX_UnloadCAs

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_UnloadCAs(WOLFSSL_CTX* ctx);

Description:
This function unloads the CA signer list and frees the whole signer table.

Return Values:
SSL_SUCCESS - returned on successful execution of the function.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.

BAD_MUTEX_E - returned if there was a mutex error. The LockMutex() did not return 0.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…

if(!wolfSSL_CTX_UnloadCAs(ctx)){
      /*The function did not unload CAs*/
}
```

See Also:
wolfSSL_CertManagerUnloadCAs
LockMutex
FreeSignerTable
UnlockMutex

wolfSSL_dtls_set_timeout_init

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_dtls_set_timeout_init(WOLFSSL* ssl, int timeout);

Description:

This function sets the dtls timeout.

Return Values:

SSL_SUCCESS - returned if the function executes without an error. The **dtls_timeout_init** and the **dtls_timeout** members of SSL have been set.

BAD_FUNC_ARG - returned if the WOLFSSL struct is NULL or if the timeout is not greater than 0. It will also return if the **timeout** argument exceeds the maximum value allowed.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

timeout - an int type that will be set to the **dtls_timeout_init** member of the WOLFSSL structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;  /*timeout value*/
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
      /*the dtls timeout was set*/
} else {
      /*Failed to set DTLS timeout. */
}
```

See Also:

wolfSSL_dtls_set_timeout_max
wolfSSL_dtls_got_timeout

wolfSSL_GetCookieCtx

Synopsis:

#include <wolfssl/ssl.h>

void* wolfSSL_GetCookieCtx(WOLFSSL* ssl);

Description:

This function returns the **IOCB_CookieCtx** member of the WOLFSSL structure.

Return Values:

The function returns a **void pointer** value stored in the IOCB_CookieCtx.

NULL - if the WOLFSSL struct is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);

if(cookie != NULL){
      /*You have the cookie */
}
```

See Also:

wolfSSL_SetCookieCtx
wolfSSL_CTX_SetGenCookie

wolfSSL_CTX_UseSessionTicket

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseSessionTicket(WOLFSSL_CTX* ctx)

Description:

This function sets wolfSSL context to use a session ticket.

Return Values:

SSL_SUCCESS: Function executed successfully.

BAD_FUNC_ARG: Returned if **ctx** is null.

MEMORY_E: Error allocating memory in internal function.

Parameters:

ctx - The WOLFSSL_CTX structure to use.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
{
    /* Error setting session ticket */
}
```

See Also:

TLSX_UseSessionTicket

wolfSSL_UseSupportedQSH

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseSupportedQSH(WOLFSSL* ssl, word16 name)

Description:

This function sets the ssl session to use supported QSH provided by name.

Return Values:

SSL_SUCCESS: Successfully set supported QSH.

BAD_FUNC_ARG: ssl is null or name is invalid.

MEMORY_E: Error allocating memory for operation.

Parameters:

ssl - Pointer to ssl session to use.

name - Name of a supported QSH. Valid names are WOLFSSL_NTRU_EESS439, WOLFSSL_NTRU_EESS593, or WOLFSSL_NTRU_EESS743.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

word16 qsh_name = WOLFSSL_NTRU_EESS439;

if(wolfSSL_UseSupportedQSH(ssl,qsh_name) != SSL_SUCCESS)
{
    /* Error setting QSH */
}
```

See Also:

TLSX_UseQSHScheme

wolfSSL_UseALPN

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseALPN(WOLFSSL* ssl, char *protocol_name_list,
    word32 protocol_name_listSz, byte options)

Description:

Setup ALPN use for a wolfSSL session.

Return Values:

SSL_SUCCESS: Success

BAD_FUNC_ARG: Returned if **ssl** or **protocol_name_list** is null or **protocol_name_listSz** is too large or **options** contain something not supported.

MEMORY_ERROR: Error allocating memory for protocol list.

SSL_FAILURE: Error

Parameters:

ssl - The wolfSSL session to use.

protocol_name_list - List of protocol names to use. Comma delimited string is required.

protocol_name_listSz - Size of the list of protocol names.

options - WOLFSSL_ALPN_CONTINUE_ON_MISMATCH or WOLFSSL_ALPN_FAILED_ON_MISMATCH.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
```

```
char alpn_list[] = { /* ALPN List */ }

if(wolfSSL_UseALPN(ssl, alpn_list, sizeof(alpn_list),
                              WOLFSSL_APN_FAILED_ON_MISMATCH) != SSL_SUCCESS)
{
    /* Error setting session ticket */
}
```

See Also:

TLSX_UseALPN

wolfSSL_CTX_trust_peer_cert

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX* ctx, const char* file,
                                                  int type);

Description:

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used.

Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT

Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

file - pointer to name of the file containing certificates

type - type of certificate being loaded ie SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);

...

ret = wolfSSL_CTX_trust_peer_cert(ctx, "./peer-cert.pem", SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
/* error loading trusted peer cert  */
}

...
```

See Also:

wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_file
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_file
wolfSSL_CTX_trust_peer_buffer

wolfSSL_CTX_Unload_trust_peers
wolfSSL_use_certificate_file
wolfSSL_use_PrivateKey_file
wolfSSL_use_certificate_chain_file

wolfSSL_CTX_trust_peer_buffer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx, const unsigned char* buffer, long sz, int type);

Description:

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file.

Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT

Please see the examples for proper usage.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

buffer - pointer to the buffer containing certificates

sz - length of the buffer input

type - type of certificate being loaded i.e. SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
// error loading trusted peer cert
}

...
```

See Also:

wolfSSL_CTX_load_verify_buffer
wolfSSL_CTX_use_certificate_file
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_CTX_use_NTRUPrivateKey_file
wolfSSL_CTX_use_certificate_chain_file
wolfSSL_CTX_trust_peer_cert
wolfSSL_CTX_Unload_trust_peers
wolfSSL_use_certificate_file

wolfSSL_use_PrivateKey_file
wolfSSL_use_certificate_chain_file

wolfSSL_CTX_Unload_trust_peers

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_Unload_trust_peers(WOLFSSL_CTX* ctx);

Description:
This function is used to unload all previously loaded trusted peer certificates.

Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG will be returned if ctx is NULL.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

Example:

```
int ret = 0;
```

```
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
// error unloading trusted peer certs
}

...
```

See Also:

wolfSSL_CTX_trust_peer_buffer
wolfSSL_CTX_trust_peer_cert

wolfSSL_CTX_allow_anon_cipher

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_allow_anon_cipher(WOLFSSL_CTX* ctx);

Description:

This function enables the **havAnon** member of the CTX structure if HAVE_ANON is defined during compilation.

Return Values:

SSL_SUCCESS - returned if the function executed successfully and the **haveAnnon** member of the CTX is set to 1.

SSL_FAILURE - returned if the CTX structure was NULL.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
```

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
#ifdef HAVE_ANON
      if(cipherList == NULL){
        wolfSSL_CTX_allow_anon_cipher(ctx);
        if(wolfSSL_CTX_set_cipher_list(ctx, "ADH_AES128_SHA") != SSL_SUCCESS)
{
            /*failure case*/
        }
      }
#endif
```

See Also:

wolfSSL_CTX_memrestore_cert_cache

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_CTX_memrestore_cert_cache(WOLFSSL* ssl);

Description:

This function restores the certificate cache from memory.

Return Values:

SSL_SUCCESS - returned if the function and subroutines executed without an error.

BAD_FUNC_ARG - returned if the **ctx** or **mem** parameters are NULL or if the **sz** parameter is less than or equal to zero.

BUFFER_E - returned if the cert cache memory buffer is too small.

CACHE_MATCH_ERROR - returned if there was a cert cache header mismatch.

BAD_MUTEX_E - returned if the lock mutex on failed.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

mem - a void pointer with a value that will be restored to the certificate cache.

sz - an int type that represents the size of the **mem** parameter.

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
void* mem;
int sz = (*int) sizeof(mem);
…
if(wolfSSL_CTX_memrestore_cert_cache(ssl->ctx, mem, sz)){
	/*The success case*/
}
```

See Also:
CM_MemRestoreCertCache

wolfSSL_CTX_SetMinVersion

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetMinVersion(WOLFSSL_CTX* ctx, int version);

Description:
This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Return Values:
SSL_SUCCESS - returned if the function returned without error and the minimum version is set.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX structure was NULL or if the minimum version is not supported.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

version - an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version;  /*macro representation */
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
      /*Failed to set min version*/
}
```

See Also:
SetMinVersionHelper

17.4 Callbacks

The functions in this section have to do with callbacks which the application is able to set in relation to wolfSSL.

wolfSSL_SetIOReadCtx

Synopsis:
void wolfSSL_SetIOReadCtx(WOLFSSL* ssl, void *rctx);

Description:
This function registers a context for the SSL session's receive callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own receive callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

Return Values:
No return values are used for this function.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

rctx - pointer to the context to be registered with the SSL session's (**ssl**) receive callback function.

Example:

```
int sockfd;
WOLFSSL* ssl = 0;
...
/*Manually setting the socket fd as the receive CTX, for example*/
wolfSSL_SetIOReadCtx(ssl, &sockfd);
...
```

See Also:
wolfSSL_SetIORecv
wolfSSL_SetIOSend
wolfSSL_SetIOWriteCtx

wolfSSL_SetIOWriteCtx

Synopsis:
void wolfSSL_SetIOWriteCtx(WOLFSSL* ssl, void *wctx);

Description:
This function registers a context for the SSL session's send callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own send callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

Return Values:
No return values are used for this function.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

wctx - pointer to the context to be registered with the SSL session's (**ssl**) send callback function.

Example:

```
int sockfd;
WOLFSSL* ssl = 0;
...
/*Manually setting the socket fd as the send CTX, for example*/
wolfSSL_SetIOSendCtx(ssl, &sockfd);
...
```

See Also:
wolfSSL_SetIORecv
wolfSSL_SetIOSend
wolfSSL_SetIOReadCtx

wolfSSL_SetIOReadFlags

Synopsis:
void wolfSSL_SetIOReadFlags(WOLFSSL* ssl, int flags);

Description:
This function sets the flags for the receive callback to use for the given SSL session. The receive callback could be either the default wolfSSL EmbedReceive callback, or a custom callback specified by the user (see wolfSSL_SetIORecv). The default flag value is set internally by wolfSSL to the value of 0.

The default wolfSSL receive callback uses the recv() function to receive data from the socket. From the recv() man page:

"The flags argument to a recv() function is formed by or'ing one or more of the values:

MSG_OOB	process out-of-band data
MSG_PEEK	peek at incoming message
MSG_WAITALL	wait for full request or error

The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned."

Return Values:
No return values are used for this function.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

flags - value of the I/O read flags for the specified SSL session (**ssl**).

Example:

```
WOLFSSL* ssl = 0;
...
/*Manually setting recv flags to 0*/
wolfSSL_SetIOReadFlags(ssl, 0);
...
```

See Also:
wolfSSL_SetIORecv
wolfSSL_SetIOSend
wolfSSL_SetIOReadCtx

wolfSSL_SetIOWriteFlags

Synopsis:
void wolfSSL_SetIOWriteFlags(WOLFSSL* ssl, int flags);

Description:
This function sets the flags for the send callback to use for the given SSL session. The send callback could be either the default wolfSSL EmbedSend callback, or a custom callback specified by the user (see wolfSSL_SetIOSend). The default flag value is set

internally by wolfSSL to the value of 0.

The default wolfSSL send callback uses the send() function to send data from the socket. From the send() man page:

“The flags parameter may include one or more of the following:

    #define MSG_OOB   0x1 /* process out-of-band data */
    #define MSG_DONTROUTE 0x4 /* bypass routing, use direct interface */

The flag MSG_OOB is used to send ``out-of-band'' data on sockets that support this notion (e.g.  SOCK_STREAM); the underlying protocol must also support ``out-of-band'' data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.”

Return Values:
No return values are used for this function.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

flags - value of the I/O send flags for the specified SSL session (**ssl**).

Example:

```
WOLFSSL* ssl = 0;
...
/*Manually setting send flags to 0*/
wolfSSL_SetIOSendFlags(ssl, 0);
...
```

See Also:
wolfSSL_SetIORecv
wolfSSL_SetIOSend
wolfSSL_SetIOReadCtx

wolfSSL_SetIORecv

Synopsis:
void wolfSSL_SetIORecv(WOLFSSL_CTX* ctx, CallbackIORecv CBIORecv);

typedef int (*CallbackIORecv)(WOLFSSL* ssl, char* buf, int sz, void* ctx);

Description:

This function registers a receive callback for wolfSSL to get input data. By default, wolfSSL uses EmbedReceive() as the callback which uses the system's TCP recv() function. The user can register a function to get input from memory, some other network module, or from anywhere. Please see the EmbedReceive() function in **src/io.c** as a guide for how the function should work and for error codes. In particular, **IO_ERR_WANT_READ** should be returned for non blocking receive when no data is ready.

Return Values:

No return values are used for this function.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

callback - function to be registered as the receive callback for the wolfSSL context, **ctx**. The signature of this function must follow that as shown above in the Synopsis section.

Example:

```
WOLFSSL_CTX* ctx = 0;

/*Receive callback prototype*/
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);

/*Register the custom receive callback with wolfSSL*/
wolfSSL_SetIORecv(ctx, MyEmbedReceive);

int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx)
{
      /*custom EmbedReceive function*/
}
```

See Also:

wolfSSL_SetIOSend
wolfSSL_SetIOReadCtx
wolfSSL_SetIOWriteCtx

wolfSSL_SetIOSend

Synopsis:

void wolfSSL_SetIOSend(WOLFSSL_CTX* ctx, CallbackIOSend CBIOSend);

typedef int (*CallbackIOSend)(WOLFSSL* ssl, char* buf, int sz, void* ctx);

Description:

This function registers a send callback for wolfSSL to write output data. By default, wolfSSL uses EmbedSend() as the callback which uses the system's TCP send() function. The user can register a function to send output to memory, some other network module, or to anywhere. Please see the EmbedSend() function in **src/io.c** as a guide for how the function should work and for error codes. In particular, **IO_ERR_WANT_WRITE** should be returned for non blocking send when the action cannot be taken yet.

Return Values:

No return values are used for this function.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

callback - function to be registered as the send callback for the wolfSSL context, **ctx**. The signature of this function must follow that as shown above in the Synopsis section.

Example:

```
WOLFSSL_CTX* ctx = 0;

/*Receive callback prototype*/
int MyEmbedSend(WOLFSSL* ssl, char* buf, int sz, void* ctx);

/*Register the custom receive callback with wolfSSL*/
wolfSSL_SetIOSend(ctx, MyEmbedSend);

int MyEmbedSend(WOLFSSL* ssl, char* buf, int sz, void* ctx)
{
      /*custom EmbedSend function*/
}
```

See Also:

wolfSSL_SetIORecv
wolfSSL_SetIOReadCtx
wolfSSL_SetIOWriteCtx

wolfSSL_CTX_set_TicketEncCb

Synopsis:
#include <wolfssl/ssl.h>

```
typedef int (*SessionTicketEncCb)(WOLFSSL*,
                        unsigned char key_name[WOLFSSL_TICKET_NAME_SZ],
                        unsigned char iv[WOLFSSL_TICKET_IV_SZ],
                        unsigned char mac[WOLFSSL_TICKET_MAC_SZ],
                        int enc, unsigned char* ticket, int inLen, int* outLen, void* userCtx);

int wolfSSL_CTX_set_TicketEncCb(WOLFSSL_CTX* ctx, SessionTicketEncCb);
```

Description:
This function sets the session ticket key encrypt callback function for a server to support session tickets as specified in RFC 5077.

Return Values:

SSL_SUCCESS will be returned upon successfully setting the session.

BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Parameters:

ctx - pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().

cb - user callback function to encrypt/decrypt session tickets

Callback Parameters:

ssl - pointer to the WOLFSSL object, created with wolfSSL_new()

key_name - unique key name for this ticket context, should be randomly generated

iv - unique IV for this ticket, up to 128 bits, should be randomly generated

mac - up to 256 bit mac for this ticket

enc - if this encrypt parameter is true the user should fill in key_name, iv, mac, and encrypt the ticket in-place of length inLen and set the resulting output length in *outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL that the encryption was successful. If this encrypt parameter is false, the user should perform a decrypt of the ticket in-place of length inLen using key_name, iv, and mac. The resulting decrypt length should be set in *outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL to proceed using the decrypted ticket. Returning WOLFSSL_TICKET_RET_CREATE tells wolfSSL to use the decrypted ticket but also to generate a new one to send to the client, helpful if recently rolled keys and don't want to force a full handshake. Returning WOLFSSL_TICKET_RET_REJECT tells wolfSSL to reject this ticket, perform a full handshake, and create a new standard session ID for normal session resumption. Returning WOLFSSL_TICKET_RET_FATAL tells wolfSSL to end the connection attempt with a fatal error.

ticket - the input/output buffer for the encrypted ticket. See the enc parameter

inLen - the input length of the ticket parameter

outLen - the resulting output length of the ticket parameter. When entering the callback outLen will indicate the maximum size available in the ticket buffer.

userCtx - the user context set with wolfSSL_CTX_set_TicketEncCtx()

Example:

```
See wolfssl/test.h myTicketEncCb() used by the example server and example
echoserver.
```

See Also:

wolfSSL_CTX_set_TicketHint
wolfSSL_CTX_set_TicketEncCtx

wolfSSL_CTX_set_TicketEncCb

Synopsis:
```
#include <wolfssl/ssl.h>

typedef int (*SessionTicketEncCb)(WOLFSSL*,
                    unsigned char key_name[WOLFSSL_TICKET_NAME_SZ],
                    unsigned char iv[WOLFSSL_TICKET_IV_SZ],
                    unsigned char mac[WOLFSSL_TICKET_MAC_SZ],
                    int enc, unsigned char* ticket, int inLen, int* outLen, void* userCtx);

int wolfSSL_CTX_set_TicketEncCb(WOLFSSL_CTX* ctx, SessionTicketEncCb);
```

Description:
This function sets the session ticket key encrypt callback function for a server to support session tickets as specified in RFC 5077.

Return Values:

SSL_SUCCESS will be returned upon successfully setting the session.

BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Parameters:

ctx - pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().

cb - user callback function to encrypt/decrypt session tickets

Callback Parameters:

ssl - pointer to the WOLFSSL object, created with wolfSSL_new()

key_name - unique key name for this ticket context, should be randomly generated

iv - unique IV for this ticket, up to 128 bits, should be randomly generated

mac - up to 256 bit mac for this ticket

enc - if this encrypt parameter is true the user should fill in key_name, iv, mac, and encrypt the ticket in-place of length inLen and set the resulting output length in *outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL that the encryption was successful. If this encrypt parameter is false, the user should perform a decrypt of the ticket in-place of length inLen using key_name, iv, and mac. The resulting decrypt length should be set in *outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL to proceed using the decrypted ticket. Returning WOLFSSL_TICKET_RET_CREATE tells wolfSSL to use the decrypted ticket but also to generate a new one to send to the client, helpful if recently rolled keys and don't want to force a full handshake. Returning WOLFSSL_TICKET_RET_REJECT tells wolfSSL to reject this ticket, perform a full handshake, and create a new standard session ID for normal session resumption. Returning WOLFSSL_TICKET_RET_FATAL tells wolfSSL to end the connection attempt with a fatal error.

ticket - the input/output buffer for the encrypted ticket. See the enc parameter

inLen - the input length of the ticket parameter

outLen - the resulting output length of the ticket parameter. When entering the callback outLen will indicate the maximum size available in the ticket buffer.

userCtx - the user context set with wolfSSL_CTX_set_TicketEncCtx()

Example:

```
See wolfssl/test.h myTicketEncCb() used by the example server and example
echoserver.
```

See Also:
wolfSSL_CTX_set_TicketHint
wolfSSL_CTX_set_TicketEncCtx

wolfSSL_CTX_set_TicketHint

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_set_TicketHint(WOLFSSL_CTX* ctx, int hint);

Description:
This function sets the session ticket hint relayed to the client. For server side use.

Return Values:

SSL_SUCCESS will be returned upon successfully setting the session.

BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Parameters:

ctx - pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().

hint - number of seconds the ticket might be valid for. Hint to client.

See Also:

wolfSSL_CTX_set_TicketEncCb()

wolfSSL_CTX_set_TicketEncCtx

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_set_TicketEncCtx(WOLFSSL_CTX* ctx, void* userCtx);

Description:

This function sets the session ticket encrypt user context for the callback. For server side use.

Return Values:

SSL_SUCCESS will be returned upon successfully setting the session.

BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Parameters:

ctx - pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().

userCtx - the user context for the callback

See Also:
wolfSSL_CTX_set_TicketEncCb()

wolfSSL_CTX_SetCACb

Synopsis:
void wolfSSL_CTX_SetCACb(WOLFSSL_CTX* ctx, CallbackCACache cb);

typedef void (*CallbackCACache)(unsigned char* der, int sz, int type);

Description:
This function registers a callback with the SSL context (WOLFSSL_CTX) to be called when a new CA certificate is loaded into wolfSSL. The callback is given a buffer with the DER-encoded certificate.

Return Values:
This function has no return value.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

callback - function to be registered as the CA callback for the wolfSSL context, **ctx**. The signature of this function must follow that as shown above in the Synopsis section.

Example:

```
WOLFSSL_CTX* ctx = 0;

/*CA callback prototype*/
int MyCACallback(unsigned char *der, int sz, int type);

/*Register the custom CA callback with the SSL context*/
wolfSSL_CTX_SetCACb(ctx, MyCACallback);
```

```
int MyCACallback(unsigned char* der, int sz, int type)
{
      /* custom CA callback function, DER-encoded cert
        located in "der" of size "sz" with type "type" */
}
```

See Also:
wolfSSL_CTX_load_verify_locations

wolfSSL_connect_ex

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_connect_ex(WOLFSSL* ssl, HandShakeCallBack hsCb,
TimeoutCallBack toCb, Timeval timeout);

typedef int (*HandShakeCallBack)(HandShakeInfo*);
typedef int (*TimeoutCallBack)(TimeoutInfo*);

typedef struct timeval Timeval;

```
typedef struct handShakeInfo_st {
     char   cipherName[MAX_CIPHERNAME_SZ + 1];  /* negotiated
                                                  name */

     char   packetNames[MAX_PACKETS_HANDSHAKE]
[MAX_PACKETNAME_SZ+1];
                                              /* SSL packet
                                                  names */
     int    numberPackets;             /* actual # of packets  */
     int    negotiationError;          /* cipher/parameter err */
} HandShakeInfo;


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
                                                   packets */
     Timeval    timeoutValue;             /* timer that caused
                                                   it */
} TimeoutInfo;


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;
```

Description:
wolfSSL_connect_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through **packetNames[]**.

The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout.

This extension can be called with either, both, or neither callbacks.

Return Values:
If successful the call will return **SSL_SUCCESS**.

GETTIME_ERROR will be returned if *gettimeofday()* encountered an error.

SETITIMER_ERROR will be returned if *setitimer()* encountered an error.

SIGACT_ERROR will be returned if *sigaction()* encountered an error.

SSL_FATAL_ERROR will be returned if the underlying *SSL_connect()* call encountered an error.

See Also:
wolfSSL_accept_ex

wolfSSL_accept_ex

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_accept_ex(WOLFSSL* ssl, HandShakeCallBack hsCb,
TimeoutCallBack toCb, Timeval timeout);

typedef int (*HandShakeCallBack)(HandShakeInfo*);
typedef int (*TimeoutCallBack)(TimeoutInfo*);

typedef struct timeval Timeval;

```
typedef struct handShakeInfo_st {
     char   cipherName[MAX_CIPHERNAME_SZ + 1];  /*negotiated
                                                  name*/


     char   packetNames[MAX_PACKETS_HANDSHAKE]
[MAX_PACKETNAME_SZ+1];
/* SSL packet names */

     int    numberPackets;              /*actual # of packets */
     int    negotiationError;          /*cipher/parameter err */
} HandShakeInfo;


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
                                                     packets */
     Timeval    timeoutValue;               /* timer that
                                                caused it*/
} TimeoutInfo;


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;
```

Description:

wolfSSL_accept_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through **packetNames[]**.

The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout.

This extension can be called with either, both, or neither callbacks.

Return Values:

If successful the call will return **SSL_SUCCESS**.

GETTIME_ERROR will be returned if *gettimeofday()* encountered an error.

SETITIMER_ERROR will be returned if *setitimer()* encountered an error.

SIGACT_ERROR will be returned if *sigaction()* encountered an error.

SSL_FATAL_ERROR will be returned if the underlying *SSL_accept()* call encountered an error.

See Also:
wolfSSL_connect_ex

wolfSSL_SetLoggingCb

Synopsis:
#include <wolfssl/wolfcrypt/logging.h>

int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

typedef void (*wolfSSL_Logging_cb)(const int logLevel, const char *const logMessage);

Description:
This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it *fprintf()* to **stderr** is used but by using this function anything can be done by the user.

Return Values:
If successful this function will return 0.

BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Parameters:

log_function - function to register as a logging callback. Function signature must follow the above prototype.

Example:

```
int ret = 0;

/*Logging callback prototype*/
void MyLoggingCallback(const int logLevel, const char* const logMessage);

/*Register the custom logging callback with wolfSSL*/
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
      /*failed to set logging callback*/
}

void MyLoggingCallback(const int logLevel, const char* const logMessage)
```

```
{
	/*custom logging function*/
}
```

See Also:
wolfSSL_Debugging_ON
wolfSSL_Debugging_OFF

wolfSSL_SetTlsHmacInner

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetTlsHmacInner(WOLFSSL* ssl, byte* inner, word32 sz,
int content, int verify);

Description:
Allows caller to set the Hmac Inner vector for message sending/receiving. The result is written to **inner** which should be at least wolfSSL_GetHmacSize() bytes. The size of the message is specified by **sz**, **content** is the type of message, and **verify** specifies whether this is a verification of a peer message. Valid for cipher types excluding **WOLFSSL_AEAD_TYPE**.

Return Values:
If successful the call will return 1.

BAD_FUNC_ARG will be returned for an error state.

See Also:
wolfSSL_GetBulkCipher()
wolfSSL_GetHmacType()

wolfSSL_CTX_SetMacEncryptCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetMacEncryptCb(WOLFSSL_CTX*, CallbackMacEncrypt);

typedef int (*CallbackMacEncrypt)(WOLFSSL* ssl, unsigned char* macOut,
    const unsigned char* macIn, unsigned int macInSz, int macContent,
    int macVerify, unsigned char* encOut, const unsigned char* encIn,
    unsigned int encSz, void* ctx);

Description:
Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback. The callback should return 0 for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **macOut** is the output buffer where the result of the mac should be stored. **macIn** is the mac input buffer and **macInSz** notes the size of the buffer. **macContent** and **macVerify** are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. **encOut** is the output buffer where the result on the encryption should be stored. **encIn** is the input buffer to encrypt while **encSz** is the size of the input. An example callback can be found wolfssl/test.h myMacEncryptCb().

Return Values:
NA

See Also:
wolfSSL_SetMacEncryptCtx()
wolfSSL_GetMacEncryptCtx()

wolfSSL_SetMacEncryptCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetMacEncryptCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetMacEncryptCb()

wolfSSL_GetMacEncryptCtx()

wolfSSL_GetMacEncryptCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetMacEncryptCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Atomic User Record Processing Mac/Encrypt Callback Context previously stored with wolfSSL_SetMacEncryptCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetMacEncryptCb()
wolfSSL_SetMacEncryptCtx()

wolfSSL_CTX_SetDecryptVerifyCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetDecryptVerifyCb(WOLFSSL_CTX*, CallbackDecryptVerify);

```
typedef int (*CallbackDecryptVerify)(WOLFSSL* ssl,
      unsigned char* decOut, const unsigned char* decIn,
      unsigned int decSz, int content, int verify, unsigned int* padSz,
      void* ctx);
```

Description:
Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback. The callback should return 0 for success or < 0 for an error. The **ssl** and **ctx** pointers are

available for the user's convenience. **decOut** is the output buffer where the result of the decryption should be stored. **decIn** is the encrypted input buffer and **decInSz** notes the size of the buffer. **content** and **verify** are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. **padSz** is an output variable that should be set with the total value of the padding. That is, the mac size plus any padding and pad bytes. An example callback can be found wolfssl/test.h myDecryptVerifyCb().

Return Values:
NA

See Also:
wolfSSL_SetMacEncryptCtx()
wolfSSL_GetMacEncryptCtx()

wolfSSL_SetDecryptVerifyCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetDecryptVerifyCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetDecryptVerifyCb()
wolfSSL_GetDecryptVerifyCtx()

wolfSSL_GetDecryptVerifyCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetDecryptVerifyCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Atomic User Record Processing Decrypt/Verify Callback Context previously stored with wolfSSL_SetDecryptVerifyCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetDecryptVerifyCb()
wolfSSL_SetDecryptVerifyCtx()

wolfSSL_CTX_SetEccSignCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetEccSignCb(WOLFSSL_CTX*, CallbackEccSign);

```
typedef int (*CallbackEccSign)(WOLFSSL* ssl,
      const unsigned char* in, unsigned int inSz,
      unsigned char* out, unsigned int* outSz,
      const unsigned char* keyDer, unsigned int keySz,
      void* ctx);
```

Description:
Allows caller to set the Public Key Callback for ECC Signing. The callback should return 0 for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **in** is the input buffer to sign while **inSz** denotes the length of the input. **out** is the output buffer where the result of the signature should be stored. **outSz** is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. **keyDer** is the ECC Private key in ASN1 format and **keySz** is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccSign().

Return Values:

NA

See Also:
wolfSSL_SetEccSignCtx()
wolfSSL_GetEccSignCtx()

wolfSSL_SetEccSignCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetEccSignCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Public Key Ecc Signing Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetEccSignCb()
wolfSSL_GetEccSignCtx()

wolfSSL_GetEccSignCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetEccSignCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetEccSignCb()
wolfSSL_SetEccSignCtx()

wolfSSL_CTX_SetEccVerifyCb

Synopsis:
#include <wolfssl/ssl.h>

```
void wolfSSL_CTX_SetEccVerifyCb(WOLFSSL_CTX*, CallbackEccVerify);

typedef int (*CallbackEccVerify)(WOLFSSL* ssl,
      const unsigned char* sig, unsigned int sigSz,
      const unsigned char* hash, unsigned int hashSz,
      const unsigned char* keyDer, unsigned int keySz,
      int* result, void* ctx);
```

Description:
Allows caller to set the Public Key Callback for ECC Verification. The callback should return 0 for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **sig** is the signature to verify and **sigSz** denotes the length of the signature. **hash** is an input buffer containing the digest of the message and **hashSz** denotes the length in bytes of the hash. **result** is an output variable where the result of the verification should be stored, **1** for success and **0** for failure. **keyDer** is the ECC Private key in ASN1 format and **keySz** is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccVerify().

Return Values:
NA

See Also:
wolfSSL_SetEccVerifyCtx()
wolfSSL_GetEccVerifyCtx()

wolfSSL_SetEccVerifyCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetEccVerifyCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Public Key Ecc Verification Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetEccVerifyCb()
wolfSSL_GetEccVerifyCtx()

wolfSSL_GetEccVerifyCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetEccVerifyCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Public Key Ecc Verification Callback Context previously stored with wolfSSL_SetEccVerifyCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetEccVerifyCb()
wolfSSL_SetEccVerifyCtx()

wolfSSL_CTX_SetRsaSignCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetEccRsaCb(WOLFSSL_CTX*, CallbackRsaSign);

```
typedef int (*CallbackRsaSign)(WOLFSSL* ssl,
      const unsigned char* in, unsigned int inSz,
      unsigned char* out, unsigned int* outSz,
      const unsigned char* keyDer, unsigned int keySz,
      void* ctx);
```

Description:
Allows caller to set the Public Key Callback for RSA Signing. The callback should return 0 for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **in** is the input buffer to sign while **inSz** denotes the length of the input. **out** is the output buffer where the result of the signature should be stored. **outSz** is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. **keyDer** is the RSA Private key in ASN1 format and **keySz** is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaSign().

Return Values:
NA

See Also:
wolfSSL_SetRsaSignCtx()
wolfSSL_GetRsaSignCtx()

wolfSSL_SetRsaSignCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetRsaSignCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Public Key RSA Signing Callback Context to **ctx**.

Return Values:

NA

See Also:
wolfSSL_CTX_SetRsaSignCb()
wolfSSL_GetRsaSignCtx()

wolfSSL_GetRsaSignCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetRsaSignCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Public Key RSA Signing Callback Context previously stored with wolfSSL_SetRsaSignCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetRsaSignCb()
wolfSSL_SetRsaSignCtx()

wolfSSL_CTX_SetRsaVerifyCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetRsaVerifyCb(WOLFSSL_CTX*, CallbackRsaVerify);

```
typedef int (*CallbackRsaVerify)(WOLFSSL* ssl,
      unsigned char* sig, unsigned int sigSz,
      unsigned char** out,
      const unsigned char* keyDer, unsigned int keySz,
```

void* ctx);

Description:
Allows caller to set the Public Key Callback for RSA Verification. The callback should return the number of plaintext bytes for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **sig** is the signature to verify and **sigSz** denotes the length of the signature. **out** should be set to the beginning of the verification buffer after the decryption process and any padding. **keyDer** is the RSA Public key in ASN1 format and **keySz** is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaVerify().

Return Values:
NA

Also:
wolfSSL_SetRsaVerifyCtx()
wolfSSL_GetRsaVerifyCtx()

wolfSSL_SetRsaVerifyCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetRsaVerifyCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Public Key RSA Verification Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetRsaVerifyCb()
wolfSSL_GetRsaVerifyCtx()

wolfSSL_GetRsaVerifyCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetRsaVerifyCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Public Key RSA Verification Callback Context previously stored with wolfSSL_SetRsaVerifyCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetRsaVerifyCb()
wolfSSL_SetRsaVerifyCtx()

wolfSSL_CTX_SetRsaEncCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetRsaEncCb(WOLFSSL_CTX*, CallbackRsaEnc);

```
typedef int (*CallbackRsaEnc)(WOLFSSL* ssl,
      const unsigned char* in, unsigned int inSz,
      unsigned char* out, unsigned int* outSz,
      const unsigned char* keyDer, unsigned int keySz,
      void* ctx);
```

Description:
Allows caller to set the Public Key Callback for RSA Public Encrypt. The callback should return 0 for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **in** is the input buffer to encrypt while **inSz** denotes the length of the input. **out** is the output buffer where the result of the encryption should be stored. **outSz** is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the encryption should be stored there before

returning. **keyDer** is the RSA Public key in ASN1 format and **keySz** is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaEnc().

Return Values:
NA

See Also:
wolfSSL_SetRsaEncCtx()
wolfSSL_GetRsaEncCtx()

wolfSSL_SetRsaEncCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetRsaEncCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Public Key RSA Public Encrypt Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetRsaEncCb()
wolfSSL_GetRsaEncCtx()

wolfSSL_GetRsaEncCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetRsaEncCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Public Key RSA Public Encrypt Callback Context previously stored with wolfSSL_SetRsaEncCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetRsaEncCb()
wolfSSL_SetRsaEncCtx()

wolfSSL_CTX_SetRsaDecCb

Synopsis:
#include <wolfssl/ssl.h>

```
void wolfSSL_CTX_SetRsaDecCb(WOLFSSL_CTX*, CallbackRsaDec);

typedef int (*CallbackRsaDec)(WOLFSSL* ssl,
      unsigned char* in, unsigned int inSz,
      unsigned char** out,
      const unsigned char* keyDer, unsigned int keySz,
      void* ctx);
```

Description:
Allows caller to set the Public Key Callback for RSA Private Decrypt. The callback should return the number of plaintext bytes for success or < 0 for an error. The **ssl** and **ctx** pointers are available for the user's convenience. **in** is the input buffer to decrypt and **inSz** denotes the length of the input. **out** should be set to the beginning of the decryption buffer after the decryption process and any padding. **keyDer** is the RSA Private key in ASN1 format and **keySz** is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaDec().

Return Values:
NA

See Also:
wolfSSL_SetRsaDecCtx()

wolfSSL_GetRsaDecCtx()

wolfSSL_SetRsaDecCtx

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetRsaDecCtx(WOLFSSL*, void* ctx);

Description:
Allows caller to set the Public Key RSA Private Decrypt Callback Context to **ctx**.

Return Values:
NA

See Also:
wolfSSL_CTX_SetRsaDecCb()
wolfSSL_GetRsaDecCtx()

wolfSSL_GetRsaDecCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetRsaDecCtx(WOLFSSL*);

Description:
Allows caller to retrieve the Public Key RSA Private Decrypt Callback Context previously stored with wolfSSL_SetRsaDecCtx().

Return Values:
If successful the call will return a valid pointer to the context.

NULL will be returned for a blank context.

See Also:
wolfSSL_CTX_SetRsaDecCb()

wolfSSL_SetRsaDecCtx()

wolfSSL_set_SessionTicket_cb

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_set_SessionTicket_cb(WOLFSSL* ssl, CallbackSessionTicket cb,
void* ctx);

Description:

This function sets the session ticket callback. The type CallbackSessionTicket is a function pointer with the signature of:

int (*CallbackSessionTicket)(WOLFSSL*, const unsigned char*, int, void*)

Return Values:

SSL_SUCCESS - returned if the function executed without error.

BAD_FUNC_ARG - returned if the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a function pointer to the type CallbackSessionTicket.

ctx - a void pointer to the session_ticket_ctx member of the WOLFSSL structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int sessionTicketCB(WOLFSSL* ssl, const unsigned char* ticket, int ticketSz,
                        void* ctx){ … }
wolfSSL_set_SessionTicket_cb(ssl, sessionTicketCB, (void*)"initial session");
```

See Also:

wolfSSL_set_SessionTicket

CallbackSessionTicket
sessionTicketCB

wolfSSL_set_session_secret_cb

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_set_session_secret_cb(WOLFSSL* ssl, SessionSecretCb cb, void* ctx);

Description:
This function sets the session secret callback function. The SessionSecretCb type has the signature:

int (*SessionSecretCb)(WOLFSSL* ssl, void* secret, int* secretSz, void* ctx).

The **sessionSecretCb** member of the WOLFSSL struct is set to the parameter **cb**.

Return Values:
SSL_SUCCESS - returned if the execution of the function did not return an error.

SSL_FATAL_ERROR - returned if the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a SessionSecretCb type that is a function pointer with the above signature.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int SessionSecretCB (WOLFSSL* ssl, void* secret, int* secretSz, void* ctx) =
SessionSecretCb;  /*Signature of SessionSecretCb*/
…
int wolfSSL_set_session_secret_cb(ssl, SessionSecretCB, (void*)ssl->ctx){
      /*Function body. */
}
```

See Also:
SessionSecretCb

wolfSSL_CTX_SetGenCookie

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_CTX_SetGenCookie(WOLFSSL_CTX* ctx, CallbackGenCookie cb);

Description:

This function sets the callback for the CBIOCookie member of the WOLFSSL_CTX structure. The CallbackGenCookie type is a function pointer and has the signature:

int (*CallbackGenCookie)(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx);

Return Values:

This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a CallbackGenCookie type function pointer with the signature of CallbackGenCookie.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int SetGenCookieCB(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx){
      /*Callback function body. */
}
…
wolfSSL_CTX_SetGenCookie(ssl->ctx, SetGenCookieCB);
```

See Also:

CallbackGenCookie

wolfSSL_SetHsDoneCb

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_SetHsDoneCb(WOLFSSL* ssl, HandShakeDoneCb cb, void* user_ctx);

Description:
This function sets the handshake done callback. The **hsDoneCb** and **hsDoneCtx** members of the WOLFSSL structure are set in this function.

Return Values:
SSL_SUCCESS - returned if the function executed without an error. The hsDoneCb and hsDoneCtx members of the WOLFSSL struct are set.

BAD_FUNC_ARG - returned if the WOLFSSL struct is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a function pointer of type HandShakeDoneCb with the signature of the form:
int (*HandShakeDoneCb)(WOLFSSL*, void*);

user_ctx - a void pointer to the user registered context.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int myHsDoneCb(WOLFSSL* ssl, void* user_ctx){
      /*callback function */
 }
…
wolfSSL_SetHsDoneCb(ssl, myHsDoneCb, NULL);
```

See Also:
HandShakeDoneCb

wolfSSL_SetFuzzerCb

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SetFuzzerCb(WOLFSSL* ssl, CallbackFuzzer cbf, void* fCtx);

Description:
This function sets the fuzzer callback.

Return Values:
This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cbf - a CallbackFuzzer type that is a function pointer of the form:
int (*CallbackFuzzer)(WOLFSSL* ssl, const unsigned char* buf, int sz,
int type, void* fuzzCtx);

fCtx - a void pointer type that will be set to the fuzzerCtx member of the WOLFSSL structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
void* fCtx;

int callbackFuzzerCB(WOLFSSL* ssl, const unsigned char* buf, int sz,
                        int type, void* fuzzCtx){
/*function definition*/
 }
…
wolfSSL_SetFuzzerCb(ssl, callbackFuzzerCB, fCtx);
```

See Also:
CallbackFuzzer

wolfSSL_CertManagerSetCRL_Cb

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerSetCRL_Cb(WOLFSSL_CERT_MANAGER* cm,
CbMissingCRL cb);

Description:
This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

Return Values:
SSL_SUCCESS - returned upon successful execution of the function and subroutines.

BAD_FUNC_ARG - returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Parameters:

cm - the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.

cb - a function pointer to (*CbMissingCRL) that is set to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url){
      /*Function body. */
}
…
CbMissingCRL cb = CbMissingCRL;
…
if(ctx){
      return wolfSSL_CertManagerSetCRL_Cb(ssl->ctx->cm, cb);
}
```

See Also:
CbMissingCRL
wolfSSL_SetCRL_Cb

wolfSSL_SetOCSP_Cb

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb, CbOCSPRespFree
respFreeCb, void* ioCbCtx);

Description:

This function sets the OCSP callback in the WOLFSSL_CERT_MANAGER structure.

Return Values:

SSL_SUCCESS - returned if the function executes without error. The ocspIOCb, ocspRespFreeCb, and ocspIOCtx memebers of the CM are set.

BAD_FUNC_ARG - returned if the WOLFSSL or WOLFSSL_CERT_MANAGER structures are NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

ioCb - a function pointer to type CbOCSPIO.

respFreeCb - a function pointer to type CbOCSPRespFree which is the call to free the response memory.

ioCbCtx - a void pointer that will be held in the ocspIOCtx member of the CM.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int OCSPIO_CB(void* , const char*, int , unsigned char* , int,
                  unsigned char**){  /*must have this signature*/
            /*Function Body*/
}
…
void OCSPRespFree_CB(void* , unsigned char* ){ /*must have this signature*/
      /*function body*/
}
…
void* ioCbCtx;
CbOCSPRespFree CB_OCSPRespFree;

if(wolfSSL_SetOCSP_Cb(ssl, OCSPIO_CB(/*Pass args*/), CB_OCSPRespFree,
```

```
                        ioCbCtx) != SSL_SUCCESS){
      /*Callback not set */
}
```

See Also:
wolfSSL_CertManagerSetOCSP_Cb
CbOCSPIO
CbOCSPRespFree

wolfSSL_SetCRL_Cb

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetCRL_Cb(WOLFSSL* ssl, CbMissingCRL cb);

Description:
Sets the CRL callback in the WOLFSSL_CERT_MANAGER structure.

Return Values:
SSL_SUCCESS - returned if the function or subroutine executes without error. The cbMissingCRL member of the WOLFSSL_CERT_MANAGER is set.

BAD_FUNC_ARG - returned if the WOLFSSL or WOLFSSL_CERT_MANAGER structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a function pointer to CbMissingCRL.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url)/*required signature */
{
      /*Function body */
}
…
```

```
int crlCb = wolfSSL_SetCRL_Cb(ssl, cb);
if(crlCb != SSL_SUCCESS){
      /*The callback was not set properly */
}
```

See Also:
CbMissingCRL
wolfSSL_CertManagerSetCRL_Cb

wolfSSL_CTX_SetOCSP_Cb

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetOCSP_Cb(WOLFSSL_CTX* ctx, CbOCSPIO ioCb,
CbOCSPRespFree respFreeCb, void* ioCbCtx);

Description:
Sets the callback for the OCSP in the WOLFSSL_CTX structure.

Return Values:
SSL_SUCCESS - returned if the function executed successfully. The ocspIOCb, ocspRespFreeCb, and ocspIOCtx members in the CM were successfully set.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX or WOLFSSL_CERT_MANAGER structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

ioCb - a CbOCSPIO type that is a function pointer.

respFreeCb - a CbOCSPRespFree type that is a function pointer.

ioCbCtx - a void pointer that will be held in the WOLFSSL_CERT_MANAGER.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
```

```
…
CbOCSPIO ocspIOCb;
CbOCSPRespFree ocspRespFreeCb;
…
void* ioCbCtx;

int isSetOCSP = wolfSSL_CTX_SetOCSP_Cb(ctx, ocspIOCb, ocspRespFreeCb,
ioCbCtx);

if(isSetOCSP != SSL_SUCCESS){
      /*The function did not return successfully. */
}
```

See Also:
wolfSSL_CertManagerSetOCSP_Cb
CbOCSPIO
CbOCSPRespFree

wolfSSL_CertManagerSetOCSP_Cb

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerSetOCSP_Cb(WOLFSSL_CERT_MANAGER* cm,
            CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void* ioCbCtx);

Description:
The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.

Return Values:
SSL_SUCCESS - returned on successful execution. The arguments are saved in the WOLFSSL_CERT_MANAGER structure.

BAD_FUNC_ARG - returned if the WOLFSSL_CERT_MANAGER is NULL.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure.

ioCb - a function pointer of type CbOCSPIO.

respFreeCb - a function pointer of type CbOCSPRespFree.

ioCbCtx - a void pointer variable to the I/O callback user registered context.

Example:

```
wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb,
                        CbOCSPRespFree respFreeCb, void* ioCbCtx){

…
return wolfSSL_CertManagerSetOCSP_Cb(ssl->ctx->cm, ioCb, respFreeCb,
ioCbCtx);
```

See Also:
wolfSSL_CertManagerSetOCSPOverrideURL
wolfSSL_CertManagerCheckOCSP
wolfSSL_CertManagerEnableOCSPStapling
wolfSSL_ENableOCSP
wolfSSL_DisableOCSP
wolfSSL_SetOCSP_Cb

wolfSSL_set_psk_client_callback

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_set_psk_client_callback(WOLFSSL* ssl, wc_psk_client_callback cb);

Description:
Sets the PSK client side callback.

Return Values:
This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a function pointer to type wc_psk_client_callback.

Example:

```
WOLFSSL* ssl;
unsigned int cb(WOLFSSL*, const char*, char*)/*Header of function*
{
      /*Funciton body */
}
…
cb = wc_psk_client_callback;
if(ssl){
      wolfSSL_set_psk_client_callback(ssl, cb);
} else {
      /*could not set callback */
}
```

See Also:
wolfSSL_CTX_set_psk_client_callback
wolfSSL_CTX_set_psk_server_callback
wolfSSL_set_psk_server_callback

wolfSSL_CTX_SetCRL_Cb

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_SetCRL_Cb(WOLFSSL_CTX* ctx, CbMissingCRL cb);

Description:
This function will set the callback argument to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER structure by calling wolfSSL_CertManagerSetCRL_Cb.

Return Values:
SSL_SUCCESS - returned for a successful execution. The WOLFSSL_CERT_MANAGER structure's member cbMssingCRL was successfully set to **cb.**

BAD_FUNC_ARG - returned if WOLFSSL_CTX or WOLFSSL_CERT_MANAGER are NULL.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

cb - a pointer to a callback function of type CbMissingCRL. Signature requirement:

void (*CbMissingCRL)(const char* url);

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
…
void cb(const char* url)/*Required signature*/
{
      /*Function body*/
}
…
if (wolfSSL_CTX_SetCRL_Cb(ctx, cb) != SSL_SUCCESS){
      /*Failure case, cb was not set correctly. */
}
```

See Also:
wolfSSL_CertManagerSetCRL_Cb
CbMissingCRL

wolfSSL_CTX_set_psk_server_callback

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_set_psk_server_callback(WOLFSSL_CTX* ctx,
                                         wc_psk_server_callback cb);

Description:
This function sets the psk callback for the server side in the WOLFSSL_CTX structure.

Return Values:
This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a function pointer for the callback and will be stored in the WOLFSSL_CTX structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
unsigned int cb(WOLFSSL*, const char*, unsigned char*, unsigned int)
/*signature requirement*/
{
	/*Function body. */
}
…
if(ctx != NULL){
wolfSSL_CTX_set_psk_server_callback(ctx, cb);
} else {
	/*The CTX object was not properly initialized. */
}
```

See Also:

wc_psk_server_callback
wolfSSL_set_psk_client_callback
wolfSSL_set_psk_server_callback
wolfSSL_CTX_set_psk_client_callback

wolfSSL_set_psk_server_callback

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_set_psk_server_callback(WOLFSSL* ssl, wc_psk_server_callback cb);

Description:

Sets the psk callback for the server side by setting the WOLFSSL structure **options** members.

Return Values:

This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

cb - a function pointer for the callback and will be stored in the WOLFSSL structure.

Example:

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
…
int cb(WOLFSSL*, const char*, unsigned char*, unsigned int)/*Required sig. */
{
	/*Function body. */
}
…
if(ssl != NULL && cb != NULL){
	wolfSSL_set_psk_server_callback(ssl, cb);
}
```

See Also:
wolfSSL_set_psk_client_callback
wolfSSL_set_psk_server_callback
wolfSSL_CTX_set_psk_server_callback
wolfSSL_CTX_set_psk_client_callback
wolfSSL_get_psk_identity_hint
wc_psk_server_callback
InitSuites

wolfSSL_CTX_set_psk_client_callback

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_set_psk_client_callback(WOLFSSL_CTX* ctx,
wc_psk_client_callback cb);

Description:
The function sets the client_psk_cb member of the WOLFSSL_CTX structure.

Return Values:
This function has no return value.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

cb - wc_psk_client_callback is a function pointer that will be stored in the WOLFSSL_CTX structure.

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*protocol def*/);
…
static WC_INLINE unsigned int my_psk_client_cb(WOLFSSL* ssl, const char*
hint,
            char* identity, unsigned int id_max_len, unsigned char* key,
            Unsigned int key_max_len){
…
wolfSSL_CTX_set_psk_client_callback(ctx, my_psk_client_cb);
```

See Also:

(*wc_psk_client_callback)
wolfSSL_set_psk_client_callback
wolfSSL_set_psk_server_callback
wolfSSL_CTX_set_psk_server_callback
wolfSSL_CTX_set_psk_client_callback

EmbedReceiveFrom

Synopsis:

#include <wolfssl/ssl.h>

int EmbedReceiveFrom(WOLFSSL* ssl, char* buf, int sz, void* ctx);

Description:

This function is the receive embedded callback.

Return Values:

This function returns the nb **bytes read** if the execution was successful.

WOLFSSL_CBIO_ERR_WANT_READ - if the connection refused or if a ‘would block’ error was thrown in the function.

WOLFSSL_CBIO_ERR_TIMEOUT - returned if the socket timed out.

WOLFSSL_CBIO_ERR_CONN_RST - returned if the connection reset.

WOLFSSL_CBIO_ERR_ISR - returned if the socket was interrupted.

WOLFSSL_CBIO_ERR_GENERAL - returned if there was a general error.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - a constant char pointer to the buffer.

sz - an int type representing the size of the buffer.

ctx - a void pointer to the WOLFSSL_CTX context.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
char* buf; /*Allocate / Initialize */
int sz = sizeof(buf)/sizeof(char);
(void*)ctx;
…

int nb = EmbedReceiveFrom(ssl, buf, sz, ctx);

if(nb > 0){
      /*nb is the number of bytes written and is positive*/
}
```

See Also:
TranslateReturnCode
RECVFROM_FUNCTION
Setsockopt

EmbedReceive

Synopsis:
#include <wolfssl/ssl.h>

int EmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);

Description:

This function is the receive embedded callback.

Return Values:

This function returns the **number of bytes** read.

WOLFSSL_CBIO_ERR_WANT_READ - returned with a “Would block” message if the last error was SOCKET_EWOULDBLCOK or SOCKET_EAGAIN.

WOLFSSL_CBIO_ERR_TIMEOUT - returned with a “Socket timeout” message.

WOLFSSL_CBIO_ERR_CONN_RST - returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.

WOLFSSL_CBIO_ERR_ISR - returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.

WOLFSSL_CBIO_ERR_WANT_READ - returned with a “Connection refused” messag if the last error was SOCKET_ECONNREFUSED.

WOLFSSL_CBIO_ERR_CONN_CLOSE - returned with a “Connection aborted” message if the last error was SOCKET_ECONNABORTED.

WOLFSSL_CBIO_ERR_GENERAL - returned with a “General error” message if the last error was not specified.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - a char pointer representation of the buffer.

sz - the size of the buffer.

ctx - a void pointer to user registered context. In the default case the ctx is a socket descriptor pointer.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf; /*The buffer. */
int sz; /* Size of buf */
void* ctx;

int bytesRead = EmbedReceive(ssl, buf, sz, ctx);
if(bytesRead <= 0){
      /*There were no bytes read. Failure case. */
}
```

See Also:
wolfSSL_dtls_get_current_timeout
TranslateReturnCode
RECV_FUNCTION

EmbedSend

Synopsis:
#include <wolfssl/ssl.h>

int EmbedSend(WOLFSSL* ssl, char* buf, int sz, void* ctx);

Description:
This function is the send embedded callback.

Return Values:
This function returns the **number of bytes** sent.

WOLFSSL_CBIO_ERR_WANT_WRITE - returned with a “Would block” message if the last error was SOCKET_EWOULDBLOCK or SOCKET_EAGAIN.

WOLFSSL_CBIO_ERR_CONN_RST - returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.

WOLFSSL_CBIO_ERR_ISR - returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.

WOLFSSL_CBIO_ERR_CONN_CLOSE - returned with a “Socket EPIPE” message if the last error was SOCKET_EPIPE.

WOLFSSL_CBIO_ERR_GENERAL - returned with a “General error” message if the last error was not specified.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - a char pointer representing the buffer.

sz - the size of the buffer.

ctx - a void pointer to user registered context.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf; /*buffer*/
int sz; /*size of buffer*/
void* ctx;

int dSent = EmbedSend(ssl, buf, sz, ctx);
if(dSent <= 0){
      /*No byes sent. Failure case. */
}
```

See Also:
TranslateReturnCode
SEND_FUNCTION
LastError
InitSSL_Ctx
LastError

EmbedSendTo

Synopsis:
#include <wolfssl/ssl.h>

int EmbedSendTo(WOLFSSL* ssl, char8 buf, int sz, void* ctx);

Description:
This function is the send embedded callback.

Return Values:
This function returns the **number of bytes** sent.

WOLFSSL_CBIO_ERR_WANT_WRITE - returned with a “Would Block” message if the last error was either SOCKET_EWOULDBLOCK or SOCKET_EAGAIN error.

WOLFSSL_CBIO_ERR_CONN_RST - returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.

WOLFSSL_CBIO_ERR_ISR - returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.

WOLFSSL_CBIO_ERR_CONN_CLOSE - returned with a “Socket EPIPE” message if the last error was WOLFSSL_CBIO_ERR_CONN_CLOSE.

WOLFSSL_CBIO_ERR_GENERAL - returned with a “General error” message if the last error was not specified.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - a char pointer representing the buffer.

sz - the size of the buffer.

ctx - a void pointer to the user registered context. The default case is a WOLFSSL_DTLS_CTX sructure.

Example:

```
WOLFSSL* ssl;
…
char* buf;
int sz; /*Size of buffer */
void* ctx;
```

```
int sEmbed = EmbedSendto(ssl, buf, sz, ctx);
if(sEmbed <= 0){
      /*No bytes sent. Failure case. */
}
```

See Also:

LastError
EmbedSend
EmbedReceive

EmbedGenerateCookie

Synopsis:

#include <wolfssl/ssl.h>

int EmbedGenerateCookie(WOLFSSL* ssl, byte* buf, int sz, void* ctx);

Description:

This function is the DTLS Generate Cookie callback.

Return Values:

This function returns the number of **bytes** copied into the buffer.

GEN_COOKIE_E - returned if the getpeername failed in EmbedGenerateCookie.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - byte pointer representing the buffer. It is the destination from XMEMCPY().

sz - the size of the buffer.

ctx - a void pointer to user registered context.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
```

```
byte buffer[BUFFER_SIZE];
int sz = sizeof(buffer)/sizeof(byte);
void* ctx;
…
int ret = EmbedGenerateCookie(ssl, buffer, sz, ctx);

if(ret > 0){
      /*EmbedGenerateCookie code block for success*/
}
```

See Also:
wc_ShaHash
EmbedGenerateCookie
XMEMCPY
XMEMSET

EmbedOcspRespFree

Synopsis:
#include <wolfssl/ssl.h>

void EmbedOcspRespFree(void* ctx, byte* resp);

Description:
This function frees the response buffer.

Return Values:
This function has no return value.

Parameters:

ctx - a void pointer to heap hint.

resp - a byte pointer representing the response.

Example:

```
void* ctx;
byte* resp; /*Response buffer. */
```

```
…
EmbedOcspRespFree(ctx, resp);
```

See Also:
XFREE

17.5 Error Handling and Debugging

The functions in this section have to do with printing and handling errors as well as enabling and disabling debugging in wolfSSL.

wolfSSL_ERR_error_string

Synopsis:
#include <wolfssl/ssl.h>

char* wolfSSL_ERR_error_string(unsigned long errNumber, char* data);

Description:
This function converts an error code returned by wolfSSL_get_error() into a more human-readable error string. **errNumber** is the error code returned by wolfSSL_get_error() and **data** is the storage buffer which the error string will be placed in.

The maximum length of **data** is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.

Return Values:
On successful completion, this function returns the same string as is returned in **data**. Upon failure, this function returns a string with the appropriate failure reason, **msg**.

Parameters:

errNumber - error code returned by wolfSSL_get_error().

data - output buffer containing human-readable error string matching **errNumber**.

Example:

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

See Also:
wolfSSL_get_error
wolfSSL_ERR_error_string_n
wolfSSL_ERR_print_errors_fp
wolfSSL_load_error_strings

wolfSSL_ERR_error_string_n

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_ERR_error_string_n(unsigned long e, char* buf, unsigned long len);

Description:
This function is a version of wolfSSL_ERR_error_string() where **len** specifies the maximum number of characters that may be written to **buf**. Like wolfSSL_ERR_error_string(), this function converts an error code returned from wolfSSL_get_error() into a more human-readable error string. The human-readable string is placed in **buf**.

Return Values:
This function has no return value.

Parameters:

e - error code returned by wolfSSL_get_error().

buff - output buffer containing human-readable error string matching **e**.

len - maximum length in characters which may be written to **buf**.

Example:

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string_n(err, buffer, 80);
printf("err = %d, %s\n", err, buffer);
```

See Also:

wolfSSL_get_error
wolfSSL_ERR_error_string
wolfSSL_ERR_print_errors_fp
wolfSSL_load_error_strings

wolfSSL_ERR_peek_last_error

Synopsis:

#include <wolfssl/ssl.h>

ERR_peek_last_error ->

unsigned long wolfSSL_ERR_peek_last_error(void);

Description:

This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

Return Values:

Returns absolute value of last error.

Parameters:

None

Example:

```
unsigned long err;
...
```

```
err = wolfSSL_ERR_peek_last_error();

// inspect err value
```

See Also:

wolfSSL_ERR_print_errors_fp

wolfSSL_ERR_print_errors_fp

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_ERR_print_errors_fp(FILE* fp, int err);

Description:

This function converts an error code returned by wolfSSL_get_error() into a more human-readable error string and prints that string to the output file - **fp**. **err** is the error code returned by wolfSSL_get_error() and **fp** is the file which the error string will be placed in.

Return Values:

This function has no return value.

Parameters:

fp - output file for human-readable error string to be written to.

err - error code returned by wolfSSL_get_error().

Example:

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

See Also:

wolfSSL_get_error

wolfSSL_ERR_error_string
wolfSSL_ERR_error_string_n
wolfSSL_load_error_strings

wolfSSL_get_error

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_get_error(WOLFSSL* ssl, int ret);

Description:

This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through **ret**.

After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human-readable error string. See wolfSSL_ERR_error_string() for more information.

Return Values:

On successful completion, this function will return the unique error code describing why the previous API function failed.

SSL_ERROR_NONE will be returned if **ret** > 0.

Parameters:

ssl - pointer to the SSL object, created with wolfSSL_new().

ret - return value of the previous function that resulted in an error return code.

Example:

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
```

```
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

See Also:

wolfSSL_ERR_error_string
wolfSSL_ERR_error_string_n
wolfSSL_ERR_print_errors_fp
wolfSSL_load_error_strings

wolfSSL_load_error_strings

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_load_error_strings(void);

Description:

This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.

Return Values:

This function has no return value.

Parameters:

This function takes no parameters.

Example:

```
wolfSSL_load_error_strings();
```

See Also:

wolfSSL_get_error
wolfSSL_ERR_error_string
wolfSSL_ERR_error_string_n
wolfSSL_ERR_print_errors_fp
wolfSSL_load_error_strings

wolfSSL_want_read

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_want_read(WOLFSSL* ssl)

Description:

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.

Return Values:

1 - wolfSSL_get_error() would return SSL_ERROR_WANT_READ, the underlying I/O has data available for reading.

0 - There is no SSL_ERROR_WANT_READ error state.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
      // underlying I/O has data available for reading (SSL_ERROR_WANT_READ)
}
```

See Also:

wolfSSL_want_write
wolfSSL_get_error

wolfSSL_want_write

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_want_write(WOLFSSL* ssl)

Description:

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.

Return Values:

1 - wolfSSL_get_error() would return SSL_ERROR_WANT_WRITE, the underlying I/O needs data to be written in order for progress to be made in the underlying SSL connection.

0 - There is no SSL_ERROR_WANT_WRITE error state.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_write(ssl);
if (ret == 1) {
      // underlying I/O needs data to be written (SSL_ERROR_WANT_WRITE)
}
```

See Also:

wolfSSL_want_read
wolfSSL_get_error

wolfSSL_Debugging_ON

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_Debugging_ON(void);

Description:
If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use *--enable-debug* or define **DEBUG_WOLFSSL**

Return Values:
If successful this function will return 0.

NOT_COMPILED_IN is the error that will be returned if logging isn't enabled for this build.

Parameters:

This function has no parameters.

Example:

```
wolfSSL_Debugging_ON();
```

See Also:
wolfSSL_Debugging_OFF
wolfSSL_SetLoggingCb

wolfSSL_Debugging_OFF

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_Debugging_OFF(void);

Description:
This function turns off runtime logging messages. If they're already off, no action is taken.

Return Values:
No return values are returned by this function.

Parameters:

This function has no parameters.

Example:

```
wolfSSL_Debugging_OFF();
```

See Also:
wolfSSL_Debugging_ON
wolfSSL_SetLoggingCb

17.6 OCSP and CRL

The functions in this section have to do with using OCSP (Online Certificate Status Protocol) and CRL (Certificate Revocation List) with wolfSSL.

wolfSSL_CTX_EnableOCSP

Synopsis:
long wolfSSL_CTX_EnableOCSP(WOLFSSL_CTX* ctx, int options);

Description:
This function sets options to configure behavior of OCSP functionality in wolfSSL. The value of **options** if formed by or'ing one or more of the following options:

WOLFSSL_OCSP_ENABLE
- enable OCSP lookups

WOLFSSL_OCSP_URL_OVERRIDE
- use the override URL instead of the URL in certificates.

The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function.

This function only sets the OCSP options when wolfSSL has been compiled with OCSP support (--enable-ocsp, #define HAVE_OCSP).

Return Values:

SSL_SUCCESS is returned upon success

SSL_FAILURE is returned upon failure

NOT_COMPILED_IN is returned when this function has been called, but OCSP support was not enabled when wolfSSL was compiled.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

options - value used to set the OCSP options.

Example:

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_options(ctx, WOLFSSL_OCSP_ENABLE);
```

See Also:
wolfSSL_CTX_OCSP_set_override_url

wolfSSL_CTX_SetOCSP_OverrideURL

Synopsis:
int wolfSSL_CTX_SetOCSP_OverrideURL(WOLFSSL_CTX* ctx, const char* url);

Description:
This function manually sets the URL for OCSP to use. By default, OCSP will use the URL found in the individual certificate unless the WOLFSSL_OCSP_URL_OVERRIDE option is set using the wolfSSL_CTX_EnableOCSP.

Return Values:

SSL_SUCCESS is returned upon success

SSL_FAILURE is returned upon failure

NOT_COMPILED_IN is returned when this function has been called, but OCSP support was not enabled when wolfSSL was compiled.

Parameters:

ctx - pointer to the SSL context, created with wolfSSL_CTX_new().

url - pointer to the OCSP URL for wolfSSL to use.

Example:

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_override_url(ctx, “custom-url-here”);
```

See Also:
wolfSSL_CTX_OCSP_set_options

wolfSSL_EnableCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_EnableCRL(WOLFSSL* ssl, int options);

Description:
Enables CRL certificate revocation.

Return Values:
SSL_SUCCESS - the function and subroutines returned with no errors.

BAD_FUNC_ARG - returned if the WOLFSSL structure is NULL.

MEMORY_E - returned if the allocation of memory failed.

SSL_FAILURE - returned if the InitCRL function does not return successfully.

NOT_COMPILED_IN - HAVE_CRL was not enabled during the compiling.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

options - an integer that is used to determine the setting of crlCheckAll member of the WOLFSSL_CERT_MANAGER structure.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_EnableCRL(ssl, WOLFSSL_CRL_CHECKALL) != SSL_SUCCESS){
      /*Failure case. SSL_SUCCESS was not returned by this function or a
subroutine */
}
```

See Also:

wolfSSL_CertManagerEnableCRL
InitCRL

wolfSSL_DisableOCSP

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_DisableOCSP(WOLFSSL* ssl);

Description:
Disables the OCSP certificate revocation option.

Return Values:
SSL_SUCCESS - returned if the function and its subroutine return with no errors. The ocspEnabled member of the WOLFSSL_CERT_MANAGER structure was successfully set.

BAD_FUNC_ARG - returned if the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(wolfSSL_DisableOCSP(ssl) != SSL_SUCCESS){
      /*Returned with an error. Failure case in this block. */
}
```

See Also:
wolfSSL_CertManagerDisableOCSP

wolfSSL_UseOCSPStapling

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_UseOCSPStapling(WOLFSSL* ssl, byte status_type, byte options);

Description:
Stapling eliminates the need to contact the CA. Stapling lowers the cost of certificate revocation check presented in OCSP.

Return Values:
SSL_SUCCESS - returned if TLSX_UseCertificateStatusRequest executes without error.

MEMORY_E - returned if there is an error with the allocation of memory.

BAD_FUNC_ARG - returned if there is an argument that has a NULL or otherwise unacceptable value passed into the function.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

status_type - a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

options - a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_UseOCSPStapling(ssl, WOLFSSL_CSR2_OCSP,
      WOLFSSL_CSR2_OCSP_USE_NONCE) != SSL_SUCCESS){

      /*Failed case. */
}
```

See Also:
TLSX_UseCertificateStatusRequest
wolfSSL_CTX_UseOCSPStapling

EmbedOcspLookup

Synopsis:
#include <wolfssl/ssl.h>

int EmbedOcspLookup(void* ctx, const char* url, int urlSz, byte* ocspReqBuf,
int ocspReqSz, byte** ocspRespBuf);

Description:
This function retrieves the OCSP response from an OCSP responder URL given an input request.

Return Values:
>0 - OCSP Response Size

-1 - Error returned.

Parameters:

ctx - a void pointer representing the heap pointer.

url - a char pointer for the OCSP url for certificate verification.

urlSz - a byte pointer for the url size.

ocspReqBuf - a byte pointer for the OCSP request buffer.

ocspReqSz - an int type representing the size of the request buffer.

ocspRespBuf - a byte pointer that holds the OCSP response.

Example:

```
WOLFSSL_CERT_MANAGER* cm;
int options;
int wolfSSL_CertManagerEnableOCSP(WOLFSSL_CERT_MANAGER* cm, int options){
…
#ifndef WOLFSSL_USER_IO
      cm->ocspIOCb = EmbedOcspLookup;
      cm->ocspRespFreeCb = EmbedOcspRespFree;
#endif
```

See Also:

Process_http_response
build_http_request
wolfSSL_CertManagerEnableOCSPStapling

wolfSSL_CTX_UseOCSPStaplingV2

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseOCSPStaplingV2(WOLFSSL_CTX* ctx, bute status_type,
byte options);

Description:

Creates and initializes the certificate status request for OCSP Stapling.

Return Values:

SSL_SUCCESS - if the function and subroutines executed without error.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX structure is NULL or if the side variable is not client side.

MEMORY_E - returned if the allocation of memory failed.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

status_type - a byte type that is located in the CertificatStatusRequest structure and must be either WOLFSSL_CSR2_OCSP or WOLFSSL_CSR2_OCSP_MULTI.

options - a byte type that will be held in CertificateStatusRequestItemV2 struct.

Example:

```
WOLFSSL_CTX* ctx  = wolfSSL_CTX_new(/*protocol method*/);
byte status_type;
byte options;
...
if(wolfSSL_CTX_UseOCSPStaplingV2(ctx, status_type, options); != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:

TLSX_UseCertificateStatusRequestV2
wc_RNG_GenerateBlock
TLSX_Push

wolfSSL_UseOCSPStaplingV2

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseOCSPStaplingV2(WOLFSSL* ssl, byte status_type, byte options);

Description:
The function sets the status type and options for OCSP.

Return Values:
SSL_SUCCESS - returned if the function and subroutines executed without error.

MEMORY_E - returned if there was an allocation of memory error.

BAD_FUNC_ARG - returned if a NULL or otherwise unaccepted argument was passed to the function or a subroutine.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

status_type - a byte type that loads the OCSP status type.

options - a byte type that holds the OCSP options, set in wolfSSL_SNI_SetOptions() and wolfSSL_CTX_SNI_SetOptions().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if (wolfSSL_UseOCSPStaplingV2(ssl, WOLFSSL_CSR2_OCSP_MULTI, 0) !=
SSL_SUCCESS){
      /*Did not execute properly. Failure case code block. */
}
```

See Also:
TLSX_UseCertificatStatusRequestV2
wolfSSL_SNI_SetOptions
wolfSSL_CTX_SNI_SetOptions

wolfSSL_CTX_LoadCRL

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_LoadCRL(WOLFSSL_CTX* ctx, const char* path, int type,
int monitor);

Description:

This function loads CRL into the WOLFSSL_CTX structure through wolfSSL_CertManagerLoadCRL().

Return Values:

SSL_SUCCESS - returned if the function and its subroutines execute without error.

BAD_FUNC_ARG - returned if this function or any subroutines are passed NULL structures.

BAD_PATH_ERROR - returned if the **path** variable opens as NULL.

MEMORY_E - returned if an allocation of memory failed.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

path - the path to the certificate.

type - an integer variable holding the type of certificate.

monitor - an integer variable used to determine if the monitor path is requested.

Example:

```
WOLFSSL_CTX* ctx;
const char* path;
…
return wolfSSL_CTX_LoadCRL(ctx, path, SSL_FILETYPE_PEM, 0);
```

See Also:

wolfSSL_CertManagerLoadCRL
LoadCRL

wolfSSL_CertManagerLoadCRLBuffer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CertManagerLoadCRLBuffer(WOLFSSL_CERT_MANAGER* cm,
const unsigned char* buff, long sz, int type);

Description:

The function loads the CRL file by calling BufferLoadCRL.

Return Values:

SSL_SUCCESS - returned if the function completed without errors.

BAD_FUNC_ARG - returned if the WOLFSSL_CERT_MANAGER is NULL .

SSL_FATAL_ERROR - returned if there is an error associated with the WOLFSSL_CERT_MANAGER.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure.

buff - a constant byte type and is the buffer.

sz - a long int representing the size of the buffer.

type - a long integer that holds the certificate type.

Example:

```
WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; /*size of buffer*/
int type;  /*cert type*/
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
      return ret;
} else {
```

```
    /*Failure case. */
}
```

See Also:
BufferLoadCRL
wolfSSL_CertManagerEnableCRL

wolfSSL_LoadCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type, int monitor);

Description:
A wrapper function that ends up calling LoadCRL to load the certificate for revocation checking.

Return Values:
WOLFSSL_SUCCESS - returned if the function and all of the subroutines executed without error.

SSL_FATAL_ERROR - returned if one of the subroutines does not return successfully.

BAD_FUNC_ARG - f the WOLFSSL_CERT_MANAGER or the WOLFSSL structure are NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

path - a constant character pointer that holds the path to the crl file.

type - an integer representing the type of certificate.

monitor - an integer variable used to verify the monitor path if requested.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* crlPemDir;
…
if(wolfSSL_LoadCRL(ssl, crlPemDir, SSL_FILETYPE_PEM, 0) != SSL_SUCCESS){
      /*Failure case. Did not return SSL_SUCCESS. */
}
```

See Also:
wolfSSL_CertManagerLoadCRL
wolfSSL_CertManagerEnableCRL
LoadCRL

wolfSSL_DisableCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_DisableCRL(WOLFSSL* ssl);

Description:
Disables CRL certificate revocation.

Return Values:
SSL_SUCCESS - wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.

BAD_FUNC_ARG - the WOLFSSL structure was NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_DisableCRL(ssl) != SSL_SUCCESS){
      /*Failure case*/
```

```
}
```

See Also:
wolfSSL_CertManagerDisableCRL
wolfSSL_CertManagerDisableOCSP

wolfSSL_CertManagerDisableOCSP

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerDisableOCSP(WOLFSSL* ssl);

Description:
Disables OCSP certificate revocation.

Return Values:
SSL_SUCCESS - wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.

BAD_FUNC_ARG - the WOLFSSL structure was NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
      /*Fail case. */
}
```

See Also:
wolfSSL_DisableCRL

wolfSSL_CertManagerCheckCRL

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CertManagerCheckCRL(WOLFSSL_CERT_MANAGER* cm, byte* der,
int sz);

Description:

Check CRL if the option is enabled and compares the cert to the CRL list.

Return Values:

SSL_SUCCESS - returns if the function returned as expected. If the crlEnabled member of the WOLFSSL_CERT_MANAGER struct is turned on.

MEMORY_E - returns if the allocated memory failed.

BAD_FUNC_ARG - if the WOLFSSL_CERT_MANAGER is NULL.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER struct.

der - pointer to a DER formatted certificate.

sz - size of the certificate.

Example:

```
WOLFSSL_CERT_MANAGER* cm;
byte* der;
int sz; /*size of der */
...
if(wolfSSL_CertManagerCheckCRL(cm, der, sz) != SSL_SUCCESS){
      /*Error returned. Deal with failure case. */
}
```

See Also:

CheckCertCRL
ParseCertRelative
wolfSSL_CertManagerSetCRL_CB

InitDecodedCert

wolfSSL_CTX_EnableCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_EnableCRL(WOLFSSL_CTX* ctx, int options);

Description:
Enables CRL certificate verification through the CTX.

Return Values:
SSL_SUCCESS - returned if this function and it's subroutines execute without errors.

BAD_FUNC_ARG - returned if the CTX struct is NULL or there was otherwise an invalid argument passed in a subroutine.

MEMORY_E - returned if there was an error allocating memory during execution of the function.

SSL_FAILURE - returned if the crl member of the WOLFSSL_CERT_MANAGER fails to initialize correctly.

NOT_COMPILED_IN - wolfSSL was not compiled with the HAVE_CRL option.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_EnableCRL(ssl->ctx, options) != SSL_SUCCESS){
```

```
    /*The function failed*/
}
```

See Also:

wolfSSL_CertManagerEnableCRL
InitCRL
wolfSSL_CTX_DisableCRL

wolfSSL_CTX_DisableCRL

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_DisableCRL(WOLFSSL_CTX* ctx);

Description:

This function disables CRL verification in the CTX structure.

Return Values:

SSL_SUCCESS - returned if the function executes without error. The crlEnabled member of the WOLFSSL_CERT_MANAGER struct is set to 0.

BAD_FUNC_ARG - returned if either the CTX struct or the CM struct has a NULL value.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);

...
if(wolfSSL_CTX_DisableCRL(ssl->ctx) != SSL_SUCCESS){
    /*Failure case.*/
}
```

See Also:

wolfSSL_CertManagerDisableCRL

wolfSSL_EnableOCSP

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_EnableOCSP(WOLFSSL* ssl, int options);

Description:
This function enables OCSP certificate verification.

Return Values:
SSL_SUCCESS - returned if the function and subroutines executes without errors.

BAD_FUNC_ARG - returned if an argument in this function or any subroutine receives an invalid argument value.

MEMORY_E - returned if there was an error allocating memory for a structure or other variable.

NOT_COMPILED_IN - returned if wolfSSL was not compiled with the HAVE_OCSP option.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

options - an integer type passed to wolfSSL_CertMangerENableOCSP() used for settings check.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int options; /*initialize to option constant*/
…
int ret = wolfSSL_EnableOCSP(ssl, options);

if(ret != SSL_SUCCESS){
      /*OCSP is not enabled*/
}
```

See Also:
wolfSSL_CertManagerEnableOCSP

wolfSSL_CTX_UseOCSPStapling

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseOCSPStapling(WOLFSSL_CTX* ctx, byte status_type,
byte options);

Description:
This function requests the certificate status during the handshake.

Return Values:
SSL_SUCCESS - returned if the function and subroutines execute without error.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX structure is NULL or otherwise if a unpermitted value is passed to a subroutine.

MEMORY_E - returned if the function or subroutine failed to properly allocate memory.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

status_type - a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

options - a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
byte statusRequest = 0; /*Initialize status request*/
…
```

```
switch(statusRequest){
      case WOLFSSL_CSR_OCSP:
            if(wolfSSL_CTX_UseOCSPStapling(ssl->ctx, WOLFSSL_CSR_OCSP,
                        WOLF_CSR_OCSP_USE_NONCE) != SSL_SUCCESS){
                        /*UseCertificateStatusRequest failed*/
                        }
                  /*Continue switch cases*/
```

See Also:
wolfSSL_UseOCSPStaplingV2
wolfSSL_UseOCSPStapling
TLSX_UseCertificateStatusRequest

wolfSSL_CTX_DisableOCSP

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_DisableOCSP(WOLFSSL_CTX* ctx);

Description:
This function disables OCSP certificate revocation checking by affecting the ocspEnabled member of the WOLFSSL_CERT_MANAGER structure.

Return Values:
SSL_SUCCESS - returned if the function executes without error. The ocspEnabled member of the CM has been disabled.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX structure is NULL.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_CTX_DisableOCSP(ssl->ctx)){
```

```
    /*OCSP is not disabled*/
}
```

See Also:
wolfSSL_DisableOCSP
wolfSSL_CertManagerDisableOCSP

wolfSSL_CTX_EnableOCSPStapling

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx);

Description:
This function enables OCSP stapling by calling wolfSSL_CertManagerEnableOCSPStapling().

Return Values:
SSL_SUCCESS - returned if there were no errors and the function executed successfully.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.

MEMORY_E - returned if there was an issue allocating memory.

SSL_FAILURE - returned if the initialization of the OCSP structure failed.

NOT_COMPILED_IN - returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

Example:

```
WOLFSSL* ssl = WOLFSSL_new();
ssl->method.version; /*set to desired protocol*/
...
if(!wolfSSL_CTX_EnableOCSPStapling(ssl->ctx)){
      /*OCSP stapling is not enabled*/
}
```

See Also:
wolfSSL_CertManagerEnableOCSPStapling
InitOCSP

wolfSSL_CertManagerEnableOCSPStapling

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CertManagerEnableOCSPStapling(WOLFSSL_CERT_MANAGER* cm);

Description:
This function turns on OCSP stapling if it is not turned on as well as set the options.

Return Values:
SSL_SUCCESS - returned if there were no errors and the function executed successfully.

BAD_FUNC_ARG - returned if the WOLFSSL_CERT_MANAGER structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.

MEMORY_E - returned if there was an issue allocating memory.

SSL_FAILURE - returned if the initialization of the OCSP structure failed.

NOT_COMPILED_IN - returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, a member of the WOLFSSL_CTX structure.

Example:

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

See Also:
wolfSSL_CTX_EnableOCSPStapling

wolfSSL_SetOCSP_OverrideURL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url);

Description:
This function sets the ocspOverrideURL member in the WOLFSSL_CERT_MANAGER structure.

Return Values:
SSL_SUCCESS - returned on successful execution of the function.

BAD_FUNC_ARG - returned if the WOLFSSL struct is NULL or if a unpermitted argument was passed to a subroutine.

MEMORY_E - returned if there was an error allocating memory in the subroutine.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

url - a constant char pointer to the url that will be stored in the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
```

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char url[URLSZ];
...
if(wolfSSL_SetOCSP_OverrideURL(ssl, url)){
      /*The override url is set to the new value*/
}
```

See Also:
wolfSSL_CertManagerSetOCSPOverrideURL

17.7 Informational

The functions in this section are informational. They allow the application to gather some kind of information about the current status or setup of wolfSSL.

wolfSSL_GetObjectSize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetObjectSize(void);

Description:
This function returns the size of the WOLFSSL object and will be dependent on build options and settings. If SHOW_SIZES has been defined when building wolfSSL, this function will also print the sizes of individual objects within the WOLFSSL object (Suites, Ciphers, etc.) to stdout.

Return Values:
This function returns the size of the WOLFSSL object.

Parameters:

This function has no parameters.

Example:

```
int size = 0;
size = wolfSSL_GetObjectSize();
```

```
printf("sizeof(WOLFSSL) = %d\n", size);
```

See Also:
wolfSSL_new();

wolfSSL_GetMacSecret

Synopsis:
#include <wolfssl/ssl.h>

const byte* wolfSSL_GetMacSecret(WOLFSSL* ssl, int verify);

Description:
Allows retrieval of the Hmac/Mac secret from the handshake process. The **verify** parameter specifies whether this is for verification of a peer message.

Return Values:
If successful the call will return a valid pointer to the secret. The size of the secret can be obtained from wolfSSL_GetHmacSize().

NULL will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

verify - specifies whether this is for verification of a peer message.

See Also:
wolfSSL_GetHmacSize()

wolfSSL_GetClientWriteKey

Synopsis:
#include <wolfssl/ssl.h>

const byte* wolfSSL_GetClientWriteKey(WOLFSSL* ssl);

Description:
Allows retrieval of the client write key from the handshake process.

Return Values:
If successful the call will return a valid pointer to the key. The size of the key can be obtained from wolfSSL_GetKeySize().

NULL will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetKeySize()
wolfSSL_GetClientWriteIV()

wolfSSL_GetClientWriteIV

Synopsis:
#include <wolfssl/ssl.h>

const byte* wolfSSL_GetClientWriteIV(WOLFSSL* ssl);

Description:
Allows retrieval of the client write IV (initialization vector) from the handshake process.

Return Values:
If successful the call will return a valid pointer to the IV. The size of the IV can be obtained from wolfSSL_GetCipherBlockSize().

NULL will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetCipherBlockSize()
wolfSSL_GetClientWriteKey()

wolfSSL_GetServerWriteKey

Synopsis:
#include <wolfssl/ssl.h>

const byte* wolfSSL_GetServerWriteKey(WOLFSSL* ssl);

Description:
Allows retrieval of the server write key from the handshake process.

Return Values:
If successful the call will return a valid pointer to the key. The size of the key can be obtained from wolfSSL_GetKeySize().

NULL will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetKeySize()
wolfSSL_GetServerWriteIV()

wolfSSL_GetServerWriteIV

Synopsis:
#include <wolfssl/ssl.h>

const byte* wolfSSL_GetServerWriteIV(WOLFSSL* ssl);

Description:
Allows retrieval of the server write IV (initialization vector) from the handshake process.

Return Values:
If successful the call will return a valid pointer to the IV. The size of the IV can be obtained from wolfSSL_GetCipherBlockSize().

NULL will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetCipherBlockSize()
wolfSSL_GetClientWriteKey()

wolfSSL_GetKeySize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetKeySize(WOLFSSL* ssl);

Description:
Allows retrieval of the key size from the handshake process.

Return Values:
If successful the call will return the key size in bytes.

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetClientWriteKey()
wolfSSL_GetServerWriteKey()

wolfSSL_GetSide

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetSide(WOLFSSL* ssl);

Description:
Allows retrieval of the side of this WOLFSSL connection.

Return Values:
If successful the call will return either **WOLFSSL_SERVER_END** or **WOLFSSL_CLIENT_END** depending on the side of WOLFSSL object.

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetClientWriteKey()
wolfSSL_GetServerWriteKey()

wolfSSL_IsTLSv1_1

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_IsTLSV1_1(WOLFSSL* ssl);

Description:
Allows caller to determine if the negotiated protocol version is at least TLS version 1.1 or greater.

Return Values:
If successful the call will return **1** for true or **0** for false.

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetSide()

wolfSSL_GetBulkCipher

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetBulkCipher(WOLFSSL* ssl);

Description:
Allows caller to determine the negotiated bulk cipher algorithm from the handshake.

Return Values:
If successful the call will return one of the following:

wolfssl_cipher_null
wolfssl_des
wolfssl_triple_des
wolfssl_aes
wolfssl_aes_gcm
wolfssl_aes_ccm
wolfssl_camellia
wolfssl_hc128
wolfssl_rabbit

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetCipherBlockSize()
wolfSSL_GetKeySize()

wolfSSL_GetCipherBlockSize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetCipherBlockSize(WOLFSSL* ssl);

Description:
Allows caller to determine the negotiated cipher block size from the handshake.

Return Values:
If successful the call will return the size in bytes of the cipher block size.

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetBulkCipher()
wolfSSL_GetKeySize()

wolfSSL_GetAeadMacSize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetAeadMacSize(WOLFSSL* ssl);

Description:
Allows caller to determine the negotiated aead mac size from the handshake. For

cipher type **WOLFSSL_AEAD_TYPE**.

Return Values:
If successful the call will return the size in bytes of the aead mac size.

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetBulkCipher()
wolfSSL_GetKeySize()

wolfSSL_GetHmacSize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetHmacSize(WOLFSSL* ssl);

Description:
Allows caller to determine the negotiated (h)mac size from the handshake. For cipher types except **WOLFSSL_AEAD_TYPE**.

Return Values:
If successful the call will return the size in bytes of the (h)mac size.

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetBulkCipher()
wolfSSL_GetHmacType()

wolfSSL_GetHmacType

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetHmacType(WOLFSSL* ssl);

Description:
Allows caller to determine the negotiated (h)mac type from the handshake. For cipher types except **WOLFSSL_AEAD_TYPE**.

Return Values:
If successful the call will return one of the following:

MD5
SHA
SHA256
SHA384

BAD_FUNC_ARG or **SSL_FATAL_ERROR** will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetBulkCipher()
wolfSSL_GetHmacSize()

wolfSSL_GetCipherType

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetCipherType(WOLFSSL* ssl);

Description:

Allows caller to determine the negotiated cipher type from the handshake.

Return Values:
If successful the call will return one of the following:

WOLFSSL_BLOCK_TYPE
WOLFSSL_STREAM_TYPE
WOLFSSL_AEAD_TYPE

BAD_FUNC_ARG will be returned for an error state.

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetBulkCipher()
wolfSSL_GetHmacType()

wolfSSL_GetOutputSize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetOutputSize(WOLFSSL* ssl, int inSz);

Description:
Returns the record layer size of the plaintext input. This is helpful when an application wants to know how many bytes will be sent across the Transport layer, given a specified plaintext input size.

This function must be called after the SSL/TLS handshake has been completed.

Return Values:
Upon success, the requested size will be returned. Upon error, one of the following will be returned:

INPUT_SIZE_E will be returned if the input size is greater than the maximum TLS fragment size (see wolfSSL_GetMaxOutputSize())

BAD_FUNC_ARG will be returned upon invalid function argument, or if the SSL/TLS handshake has not been completed yet

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

inSz - size of plaintext data

See Also:
wolfSSL_GetMaxOutputSize()

wolfSSL_GetMaxOutputSize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetMaxOutputSize(WOLFSSL* ssl);

Description:
Returns the maximum record layer size for plaintext data. This will correspond to either the maximum SSL/TLS record size as specified by the protocol standard, the maximum TLS fragment size as set by the TLS Max Fragment Length extension.

This function is helpful when the application has called wolfSSL_GetOutputSize() and received a INPUT_SIZE_E error.

This function must be called after the SSL/TLS handshake has been completed.

Return Values:
Upon success, the maximum output size will be returned. Upon error, one of the following will be returned:

BAD_FUNC_ARG will be returned upon invalid function argument, or if the SSL/TLS handshake has not been completed yet

Parameters:

ssl - a pointer to a WOLFSSL object, created using wolfSSL_new().

See Also:
wolfSSL_GetOutputSize()

17.8 Connection, Session, and I/O

The functions in this section deal with setting up the SSL/TLS connection, managing SSL sessions, and input/output.

wolfSSL_accept

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_accept(WOLFSSL* ssl);

Description:
This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up.

wolfSSL_accept() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

Return Values:
If successful the call will return **SSL_SUCCESS**.

SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
      err = wolfSSL_get_error(ssl, ret);
      printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

See Also:
wolfSSL_get_error
wolfSSL_connect

wolfSSL_connect

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_connect(WOLFSSL* ssl);

Description:
This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up.

wolfSSL_connect() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred.

wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (-155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling:

SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0);

before calling SSL_new(); Though it's not recommended.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_connect(ssl);
if (ret != SSL_SUCCESS) {
      err = wolfSSL_get_error(ssl, ret);
      printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

See Also:

wolfSSL_get_error
wolfSSL_accept

wolfSSL_connect_cert

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_connect_cert(WOLFSSL* ssl);

Description:

This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up.

wolfSSL_connect_cert() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_connect_cert() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect_cert() to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_FAILURE will be returned if the SSL session parameter is NULL.

SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_connect_cert(ssl);
if (ret != SSL_SUCCESS) {
      err = wolfSSL_get_error(ssl, ret);
      printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

See Also:
wolfSSL_get_error
wolfSSL_connect
wolfSSL_accept

wolfSSL_get_fd

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_fd(const WOLFSSL* ssl);

Description:
This function returns the file descriptor (**fd**) used as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.

Return Values:
If successful the call will return the SSL session file descriptor.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

See Also:
wolfSSL_set_fd

wolfSSL_get_session

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_SESSION* wolfSSL_get_session(WOLFSSL* ssl);

Description:
This function returns a pointer to the current session (WOLFSSL_SESSION) used in **ssl**. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake.

For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default.

Return Values:
If successful the call will return a pointer to the the current SSL session object.

NULL will be returned if **ssl** is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
WOLFSSL* ssl = 0;
WOLFSSL_SESSION* session = 0;
...
```

```
session = wolfSSL_get_session(ssl);
if (session == NULL) {
      /*failed to get session pointer*/
}
...
```

See Also:
wolfSSL_set_session

wolfSSL_get_using_nonblock

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_using_nonblock(WOLFSSL* ssl);

Description:
This function allows the application to determine if wolfSSL is using non-blocking I/O. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0.

After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Return Values:

0 - underlying I/O is blocking.

1 - underlying I/O is non-blocking.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
      /*underlying I/O is non-blocking*/
```

```
}
...
```

See Also:
wolfSSL_set_session

wolfSSL_flush_sessions

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_flush_sessions(WOLFSSL_CTX *ctx, long tm);

Description:
This function flushes session from the session cache which have expired. The time, **tm**, is used for the time comparison.

Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.

Return Values:
This function does not have a return value.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

tm - time used in session expiration comparison.

Example:

```
WOLFSSL_CTX* ssl;
...

wolfSSL_flush_sessions(ctx, time(0));
```

See Also:
wolfSSL_get_session
wolfSSL_set_session

wolfSSL_negotiate

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_negotiate(WOLFSSL* ssl);

Description:
Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an *wolfSSL_connect()* is done while a *wolfSSL_accept()* is performed if called from the server side.

Return Values:
SSL_SUCCESS will be returned if successful. (Note, older versions will return 0.)

SSL_FATAL_ERROR will be returned if the underlying call resulted in an error. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:
```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
if (ret == SSL_FATAL_ERROR) {
      /*SSL establishment failed*/
      int error_code = wolfSSL_get_error(ssl);
      ...
}
...
```

See Also:
SSL_connect
SSL_accept

wolfSSL_peek

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_peek(WOLFSSL* ssl, void* data, int sz);

Description:
This function copies **sz** bytes from the SSL session (**ssl**) internal read buffer into the buffer **data**. This function is identical to wolfSSL_read() except that the data in the internal SSL session receive buffer is not removed or modified.

If necessary, like wolfSSL_read(), wolfSSL_peek() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept().

The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in <wolfssl_root>/wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_peek() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not-yet-decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_peek() / wolfSSL_read().

If **sz** is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.

Return Values:

>0 - the number of bytes read upon success.

0 - will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.

SSL_FATAL_ERROR - will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_peek() again. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

data - buffer where wolfSSL_peek() will place data read.

sz - number of bytes to read into **data**.

Example:

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
      /*"input" number of bytes returned into buffer "reply"*/
}
```

See Also:
wolfSSL_read

wolfSSL_pending

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_pending(WOLFSSL* ssl);

Description:
This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().

Return Values:

This function returns the number of bytes pending.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

Example:

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

See Also:
wolfSSL_recv
wolfSSL_read
wolfSSL_peek

wolfSSL_read

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_read(WOLFSSL* ssl, void* data, int sz);

Description:
This function reads **sz** bytes from the SSL session (**ssl**) internal read buffer into the buffer **data**. The bytes read are removed from the internal receive buffer.

If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept().

The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in <wolfssl_root>/wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not-yet-decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read().

If **sz** is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.

Return Values:

>0 - the number of bytes read upon success.

0 - will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.

SSL_FATAL_ERROR - will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_read() again. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

data - buffer where wolfSSL_read() will place data read.

sz - number of bytes to read into **data**.

Example:

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
      /*"input" number of bytes returned into buffer "reply"*/
}
```

See wolfSSL examples (client, server, echoclient, echoserver) for more complete examples of wolfSSL_read().

See Also:
wolfSSL_recv
wolfSSL_write
wolfSSL_peek
wolfSSL_pending

wolfSSL_recv

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_recv(WOLFSSL* ssl, void* data, int sz, int flags);

Description:
This function reads **sz** bytes from the SSL session (**ssl**) internal read buffer into the buffer **data** using the specified **flags** for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read() except that it allows the application to set the recv flags for the underlying read operation.

If necessary wolfSSL_recv() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept().

The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in <wolfssl_root>/wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not-yet-decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv().

If **sz** is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.

Return Values:

>0 - the number of bytes read upon success.

0 - will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.

SSL_FATAL_ERROR - will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_recv() again. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

data - buffer where wolfSSL_recv() will place data read.

sz - number of bytes to read into **data**.

flags - the recv flags to use for the underlying recv operation.

Example:

```
WOLFSSL* ssl = 0;
char reply[1024];
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
      /*"input" number of bytes returned into buffer "reply"*/
}
```

See Also:
wolfSSL_read
wolfSSL_write
wolfSSL_peek
wolfSSL_pending

wolfSSL_send

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_send(WOLFSSL* ssl, const void* data, int sz, int flags);

Description:

This function writes **sz** bytes from the buffer, **data**, to the SSL connection, **ssl**, using the specified **flags** for the underlying write operation.

If necessary wolfSSL_send() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept().

wolfSSL_send() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_send() will return when the underlying I/O could not satisfy the needs of wolfSSL_send to continue. In this case, a call to wolfSSL_get_error() will yield either **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready.

If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer **data** of size **sz** has been completely written or an error occurred.

Return Values:

>0 - the number of bytes written upon success.

0 - will be returned upon failure. Call wolfSSL_get_error() for the specific error code.

SSL_FATAL_ERROR - will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_send() again. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

data - data buffer to send to peer.

sz - size, in bytes, of **data** to be sent to peer.

flags - the send flags to use for the underlying send operation.

Example:

```
WOLFSSL* ssl = 0;
char msg[64] = "hello wolfssl!";
int msgSz = (int)strlen(msg);
int flags = ... ;
...

input = wolfSSL_send(ssl, msg, msgSz, flags);
if (input != msgSz) {
      // wolfSSL_send() failed
}
```

See Also:
wolfSSL_write
wolfSSL_read
wolfSSL_recv

wolfSSL_write

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_write(WOLFSSL* ssl, const void* data, int sz);

Description:
This function writes **sz** bytes from the buffer, **data**, to the SSL connection, **ssl**.

If necessary, wolfSSL_write() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept().

wolfSSL_write() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_write() will return when the underlying I/O could not satisfy the needs of wolfSSL_write() to continue. In this case, a call to wolfSSL_get_error() will yield either **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready.

If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer **data** of size **sz** has been completely written or an error occurred.

Return Values:

>0 - the number of bytes written upon success.

0 - will be returned upon failure. Call wolfSSL_get_error() for the specific error code.

SSL_FATAL_ERROR - will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() again. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

data - data buffer which will be sent to peer.

sz - size, in bytes, of data to send to the peer (**data**).

Example:

```
WOLFSSL* ssl = 0;
char msg[64] = "hello wolfssl!";
int msgSz = (int)strlen(msg);
int flags;
int ret;
...

ret = wolfSSL_write(ssl, msg, msgSz);
if (ret <= 0) {
      /*wolfSSL_write() failed, call wolfSSL_get_error()*/
}
```

See wolfSSL examples (client, server, echoclient, echoserver) for more more detailed examples of wolfSSL_write().

See Also:
wolfSSL_send
wolfSSL_read
wolfSSL_recv

wolfSSL_writev

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_writev(WOLFSSL* ssl, const struct iovec* iov, int iovcnt);

Description:
Simulates writev semantics but doesn’t actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.

Return Values:

>0 - the number of bytes written upon success.

0 - will be returned upon failure. Call wolfSSL_get_error() for the specific error code.

MEMORY_ERROR will be returned if a memory error was encountered.

SSL_FATAL_ERROR - will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() again. Use wolfSSL_get_error() to get a specific error code.

Parameters:

ssl - pointer to the SSL session, created with wolfSSL_new().

iov - array of I/O vectors to write

iovcnt - number of vectors in **iov** array.

Example:

```
WOLFSSL* ssl = 0;
char *bufA = “hello\n”;
char *bufB = “hello world\n”;
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = buffA;
iov[0].iov_len = strlen(buffA);
iov[1].iov_base = buffB;
```

```
iov[1].iov_len = strlen(buffB);
iovcnt = 2;
...

ret = wolfSSL_writev(ssl, iov, iovcnt);
/*wrote "ret" bytes, or error if <= 0.*/
```

See Also:

wolfSSL_write

wolfSSL_SESSION_get_peer_chain

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_X509_CHAIN*
wolfSSL_SESSION_get_peer_chain(WOLFSSL_SESSION* session);

Description:

Returns the peer certificate chain from the WOLFSSL_SESSION struct.

Return Values:

A **pointer** to a WOLFSSL_X509_CHAIN structure that contains the peer certification chain.

Parameters:

session - a pointer to a WOLFSSL_SESSION structure.

Example:

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
	/*There was no chain. Failure case. */
}
```

See Also:

get_locked_session_stats
wolfSSL_GetSessionAtIndex

wolfSSL_GetSessionIndex
AddSession

wolfSSL_get_session_cache_memsize

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_session_cache_memsize(void);

Description:
This function returns how large the session cache save buffer should be.

Return Values:
This function returns an **integer** that represents the size of the session cache save buffer.

Parameters:

This function has no parameters.

Example:

```
int sz = /*Minimum size for error checking*/;
...
if(sz < wolfSSL_get_session_cache_memsize()){
      /*Memory buffer is too small*/
}
```

See Also:
wolfSSL_memrestore_session_cache

wolfSSL_set_SessionTicket

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_set_SessionTicket(WOLFSSL* ssl, byte* buf, word32 bufSz);

Description:
This function sets the **ticket** member of the **WOLFSSL_SESSION** structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.

Return Values:
SSL_SUCCESS - returned on successful execution of the function. The function returned without errors.

BAD_FUNC_ARG - returned if the WOLFSSL structure is NULL. This will also be thrown if the **buf** argument is NULL but the **bufSz** argument is not zero.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - a byte pointer that gets loaded into the ticket member of the session structure.

bufSz - a word32 type that represents the size of the buffer.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; /*File to load*/
word32 bufSz; /*size of buffer*/
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
      /*There was an error loading the buffer to memory. */
}
```

See Also:
wolfSSL_set_SessionTicket_cb

nwolfSSL_GetSessionAtIndex

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetSessionAtIndex(int idx, WOLFSSL_SESSION* session);

Description:
This function gets the session at specified index of the session cache and copies it into

memory. The WOLFSSL_SESSION structure holds the session information.

Return Values:
SSL_SUCCESS - returned if the function executed successfully and no errors were thrown.

BAD_MUTEX_E - returned if there was an unlock or lock mutex error.

SSL_FAILURE - returned if the function did not execute successfully.

Parameters:

idx - an int type representing the session index.

session - a pointer to the WOLFSSL_SESSION structure.

Example:

```
int idx; /*The index to locate the session. */
WOLFSSL_SESSION* session;  /*Buffer to copy to. */
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:
UnLockMutex
LockMutex
wolfSSL_GetSessionIndex

wolfSSL_GetSessionIndex

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_GetSessionIndex(WOLFSSL* ssl);

Description:
This function gets the session index of the WOLFSSL structure.

Return Values:
The function returns an int type representing the **sessionIndex** within the WOLFSSL struct.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int sesIdx = wolfSSL_GetSessionIndex(ssl);

if(sesIdx < 0 || sesIdx > sizeof(ssl->sessionIndex)/sizeof(int)){
      /* You have an out of bounds index number and something is not
            right. */
}
```

See Also:
wolfSSL_GetSessionAtIndex

wolfSSL_save_session_cache

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_save_session_cache(const char* fname);

Description:
This function persists the session cache to file. It doesn't use memsave because of additional memory use.

Return Values:
SSL_SUCCESS - returned if the function executed without error. The session cache has been written to a file.

SSL_BAD_FILE - returned if **fname** cannot be opened or is otherwise corrupt.

FWRITE_ERROR - returned if XFWRITE failed to write to the file.

BAD_MUTEX_E - returned if there was a mutex lock failure.

Parameters:

fname - is a constant char pointer that points to a file for writing.

Example:

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
      /*Fail to write to file. */
}
```

See Also:
XFWRITE
wolfSSL_restore_session_cache
wolfSSL_memrestore_session_cache

wolfSSL_memrestore_session_cache

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_memrestore_session_cache(const void* mem, int sz);

Description:
This function restores the persistent session cache from memory.

Return Values:
SSL_SUCCESS - returned if the function executed without an error.

BUFFER_E - returned if the memory buffer is too small.

BAD_MUTEX_E - returned if the session cache mutex lock failed.

CACHE_MATCH_ERROR - returned if the session cache header match failed.

Parameters:

mem - a constant void pointer containing the source of the restoration.

sz - an integer representing the size of the memory buffer.

Example:

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
      /*Failure case. SSL_SUCCESS was not returned. */
}
```

See Also:

wolfSSL_save_session_cache

wolfSSL_PrintSessionStats

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_PrintSessionStats(void);

Description:

This function prints the statistics from the session.

Return Values:

SSL_SUCCESS - returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.

BAD_FUNC_ARG - returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.

BAD_MUTEX_E - returned if there was a mutex error in the subroutine.

Parameters:
This function takes no parameters.

Example:

```
/*You will need to have a session object to retrieve stats from. */
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS     ){
      /*Did not print session stats*/
      }
```

See Also:
wolfSSL_get_session_stats

wolfSSL_restore_session_cache

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_restore_session_cache(WOLFSSL* ssl);

Description:
This function restores the persistent session cache from file. It does not use memstore because of additional memory use.

Return Values:
SSL_SUCCESS - returned if the function executed without error.

SSL_BAD_FILE - returned if the file passed into the function was corrupted and could not be opened by XFOPEN.

FREAD_ERROR - returned if the file had a read error from XFREAD.

CACHE_MATCH_ERROR - returned if the session cache header match failed.

BAD_MUTEX_E - returned if there was a mutex lock failure.

Parameters:

fname - a constant char pointer file input that will be read.

Example:

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
/*Failure case. The function did not return SSL_SUCCESS. */
}
```

See Also:
XFREAD
XFOPEN

wolfSSL_get_session_stats

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_session_stats(word32* active, word32 *total, word32* peak,
word32* maxSessions);

Description:
This function gets the statistics for the session.

Return Values:
SSL_SUCCESS - returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.

BAD_FUNC_ARG - returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.

BAD_MUTEX_E - returned if there was a mutex error in the subroutine.

Parameters:

active - a word32 pointer representing the total current sessions.

total - a word32 pointer representing the total sessions.

peak - a word32 pointer representing the peak sessions.

maxSessions - a word32 pointer representing the maximum sessions.

Example:

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow, &totalSessionsSeen, &peak,
                                    &maxSessions);
…
return ret;
```

See Also:
get_locked_session_stats
wolfSSL_PrintSessionStats

wolfSSL_session_reused

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_session_reused(WOLFSSL* ssl);

Description:
This function returns the **resuming** member of the **options** struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.

Return Values:
This function returns an int type held in the **Options** structure representing the flag for session reuse.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(!wolfSSL_session_reused(sslResume)){
      /*No session reuse allowed. */
}
```

See Also:

wolfSSL_SESSION_free
wolfSSL_GetSessionIndex
wolfSSL_memsave_session_cache

wolfSSL_memsave_session_cache

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_memsave_session_cache(void* mem, int sz);

Description:
This function persists session cache to memory.

Return Values:
SSL_SUCCESS - returned if the function executed without error. The session cache has been successfully persisted to memory.

BAD_MUTEX_E - returned if there was a mutex lock error.

BUFFER_E - returned if the buffer size was too small.

Parameters:

mem - a void pointer representing the destination for the memory copy, XMEMCPY().

sz - an int type representing the size of **mem**.

Example:

```
void* mem;
int sz; /*Max size of the memory buffer. */
…
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
      /*Failure case, you did not persist the session cache to memory */
}
```

See Also:
XMEMCPY
wolfSSL_get_session_cache_memsize

wolfSSL_SetIO_NetX

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_SetIO_NetX(WOLFSSL* ssl, NX_TCP_SOCKET* nxSocket,
ULONG waitOption);

Description:

This function sets the **nxSocket** and **nxWait** members of the **nxCtx** struct within the WOLFSSL structure.

Return Values:

This function has no return value.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

nxSocket - a pointer to type NX_TCP_SOCKET that is set to the **nxSocket** member of the **nxCTX** structure.

waitOption - a ULONG type that is set to the **nxWait** member of the **nxCtx** structure.

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_TCP_SOCKET* nxSocket; /*Initialize */
ULONG waitOption; /*Initialize */
…
if(ssl != NULL || nxSocket != NULL || waitOption <= 0){
      wolfSSL_SetIO_NetX(ssl, nxSocket, waitOption);
} else {
      /*You need to pass in good parameters. */
}
```

See Also:

set_fd
NetX_Send
NetX_Receive

wolfSSL_GetIOReadCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetIOReadCtx(WOLFSSL* ssl);

Description:
This function returns the IOCB_ReadCtx member of the WOLFSSL struct.

Return Values:
This function returns a void pointer to the **IOCB_ReadCtx** member of the WOLFSSL structure.

NULL - returned if the WOLFSSL struct is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
      /*Failure case. The ssl object was NULL. */
}
```

See Also:
wolfSSL_GetIOWriteCtx
wolfSSL_SetIOReadFlags
wolfSSL_SetIOWriteCtx
wolfSSL_SetIOReadCtx
wolfSSL_SetIOSend

wolfSSL_GetIOWriteCtx

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_GetIOWriteCtx(WOLFSSL* ssl);

Description:
This function returns the IOCB_WriteCtx member of the WOLFSSL structure.

Return Values:
This function returns a void pointer to the **IOCB_WriteCtx** member of the WOLFSSL structure.

NULL - returned if the WOLFSSL struct is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
      /*The funciton returned NULL. */
}
```

See Also:
wolfSSL_GetIOReadCtx
wolfSSL_SetIOWriteCtx
wolfSSL_SetIOReadCtx
wolfSSL_SetIOSend

wolfSSL_Rehandshake

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_Rehandshake(WOLFSSL* ssl);

Description:

This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.

Return Values:

SSL_SUCCESS - returned if the function executed without error.

BAD_FUNC_ARG - returned if the WOLFSSL structure was NULL or otherwise if an unacceptable argument was passed in a subroutine.

SECURE_RENEGOTIATION_E - returned if there was an error with renegotiating the handshake.

SSL_FATAL_ERROR - returned if there was an error with the server or client configuration and the renegotiation could not be completed. See wolfSSL_negotiate().

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
      /*There was an error and the rehandshake is not successful. */
 }
```

See Also:

wolfSSL_negotiate
wc_InitSha512
wc_InitSha384
wc_InitSha256
wc_InitSha
wc_InitMd5

wolfSSL_UseSecureRenegotiation

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseSecureRenegotiation(WOLFSSL* ssl)

Description:

This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.

Return Values:

SSL_SUCCESS: Successfully set secure renegotiation.

BAD_FUNC_ARG: Returns error if ssl is null.

MEMORY_E: Returns error if unable to allocate memory for secure renegotiation.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    /* Error setting secure renegotiation */
}
```

See Also:

TLSX_Find
TLSX_UseSecureRenegotiation

wolfSSL_UseSessionTicket

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseSessionTicket(WOLFSSL* ssl)

Description:

Force provided **WOLFSSL** structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.

Return Values:

SSL_SUCCESS: Successfully set use session ticket.

BAD_FUNC_ARG: Returned if *ssl* is null.

MEMORY_E: Error allocating memory for setting session ticket.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)
{
    /* Error setting session ticket */
}
```

See Also:

TLSX_UseSessionTicket

wolfSSL_get_current_cipher_suite

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_get_current_cipher_suite(WOLFSSL* ssl)

Description:

Returns the current cipher suit an ssl session is using.

Return Values:

ssl->options.cipherSuite: An integer representing the current cipher suite.

0: The ssl session provided is null.

Parameters:

ssl - The SSL session to check.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_get_current_cipher_suite(ssl) == 0)
{
    /* Error getting cipher suite */
}
```

See Also:

wolfSSL_CIPHER_get_name
wolfSSL_get_current_cipher
wolfSSL_get_cipher_list

wolfSSL_get_cipher_list

Synopsis:

#include <wolfssl/ssl.h>

char* wolfSSL_get_cipher_list(int priority)

Description:

Get the name of cipher at priority level passed in.

Return Values:

string: Success

0: Priority is either out of bounds or not valid.

Parameters:

priority - Integer representing the priority level of a cipher.

Example:

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

See Also:

wolfSSL_CIPHER_get_name
wolfSSL_get_current_cipher

wolfSSL_isQSH

Synopsis:

#include <wolfssl/ssl.h>

wolfSSL_isQSH(WOLFSSL* ssl)

Description:

Checks if QSH is used in the supplied SSL session.

Return Values:

0: Not used

1: Is used

Parameters:

ssl - Pointer to the SSL session to check.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_isQSH(ssl) == 1)
{
    /* SSL is using QSH. */
}
```

See Also:

wolfSSL_UseSupportedQSH

wolfSSL_get_version

Synopsis:

```
#include <wolfssl/ssl.h>
const char* wolfSSL_get_version(WOLFSSL* ssl)
```

Description:

Returns the SSL version being used as a string.

Return Values:

"SSLv3": Using SSLv3

"TLSv1": Using TLSv1

"TLSv1.1": Using TLSv1.1

"TLSv1.2": Using TLSv1.2

"TLSv1.3": Using TLSv1.3

"DTLS": Using DTLS

"DTLSv1.2": Using DTLSv1.2

"unknown": There was a problem determining which version of TLS being used.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

printf(wolfSSL_get_version("Using version: %s", ssl));
```

See Also:

wolfSSL_lib_version

wolfSSL_get_ciphers

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_get_ciphers(char* buf, int len);

Description:

This function gets the ciphers enabled in wolfSSL.

Return Values:

SSL_SUCCESS - returned if the function executed without error.

BAD_FUNC_ARG - returned if the **buf** parameter was NULL or if the **len** argument was less than or equal to zero.

BUFFER_E - returned if the buffer is not large enough and will overflow.

Parameters:

buf - a char pointer representing the buffer.

len - the length of the buffer.

Example:

```
static void ShowCiphers(void){
	char* ciphers;  /*initialize*/
	int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

	if(ret == SSL_SUCCES){
		printf("%s\n", ciphers);
	}
}
```

See Also:

GetCipherNames
wolfSSL_get_cipher_list
ShowCiphers

wolfSSL_get_verify_depth

Synopsis:
#include <wolfssl/ssl.h>

long wolfSSL_get_verify_depth(WOLFSSL* ssl);

Description:
This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non-null session object (ssl).

Return Values:
MAX_CHAIN_DEPTH - returned if the WOLFSSL_CTX structure is not NULL. By default the value is 9.

BAD_FUNC_ARG - returned if the WOLFSSL_CTX structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
      /*The verified depth is greater than what was expected*/
} else {
      /*The verified depth is smaller or equal to the expected value */
}
```

See Also:
wolfSSL_CTX_get_verify_depth

wolfSSL_get_cipher

Synopsis:
#include <wolfssl/ssl.h>

const char* wolfSSL_get_cipher(WOLFSSL* ssl);

Description:
This function matches the cipher suite in the SSL object with the available suites.

Return Values:
This function returns the **string** value of the suite matched. It will return “None” if there are no suites matched.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
#ifdef WOLFSSL_DTLS
…

/*make sure a valid suite is used */
if(wolfSSL_get_cipher(ssl) == NULL){
      WOLFSSL_MSG(“Can not match cipher suite imported”);
      return MATCH_SUITE_ERROR;
}
…
#endif /*WOLFSSL_DTLS */
```

See Also:
wolfSSL_CIPHER_get_name
wolfSSL_get_current_cipher

wolfSSL_CIPHER_get_name

Synopsis:
#include <wolfssl/ssl.h>

const char* wolfSSL_CIPHER_get_name(const WOLFSSL_CIPHER* cipher);

Description:
This function matches the cipher suite in the SSL object with the available suites and returns the string representation.

Return Values:

This function returns the **string** representation of the matched cipher suite. It will return "None" if there are no suites matched.

Parameters:

cipher - a constant pointer to a WOLFSSL_CIPHER structure.

Example:

```
WOLFSSL* ssl;

/*gets cipher name in the format DHE_RSA ...*/
const char* wolfSSL_get_cipher_name_internal(WOLFSSL* ssl){
      WOLFSSL_CIPHER* cipher;
      const char* fullName;
…
      cipher = wolfSSL_get_curent_cipher(ssl);
      fullName = wolfSSL_CIPHER_get_name(cipher);

      if(fullName){
            /*sanity check on returned cipher*/
      }
```

See Also:

wolfSSL_get_cipher
wolfSSL_get_current_cipher
wolfSSL_get_cipher_name_internal
wolfSSL_get_cipher_name

wolfSSL_get_cipher_name

Synopsis:

#include <wolfssl/ssl.h>

const char* wolfSSL_get_cipher_name(WOLFSSL* ssl);

Description:

This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.

Return Values:

This function returns the **string** representation of the cipher suite that was matched.

NULL - error or cipher not found.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
char* cipherS = wolfSSL_get_cipher_name(ssl);

if(cipher == NULL){
      /*There was not a cipher suite matched */
} else {
      /*There was a cipher suite matched*/
      printf("%s\n", cipherS);
}
```

See Also:

wolfSSL_CIPHER_get_name
wolfSSL_get_current_cipher
wolfSSL_get_cipher_name_internal

wolfSSL_get_current_cipher

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_CIPHER* wolfSSL_get_current_cipher(WOLFSSL* ssl);

Description:

This function returns a pointer to the current cipher in the ssl session.

Return Values:

The function returns the **address** of the cipher member of the WOLFSSL struct. This is a pointer to the WOLFSSL_CIPHER structure.

NULL - returned if the WOLFSSL structure is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
      /*Failure case. */
} else {
      /*The cipher was returned to cipherCurr */
}
```

See Also:

wolfSSL_get_cipher
wolfSSL_get_cipher_name_internal
wolfSSL_get_cipher_name

wolfSSL_get_SessionTicket

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_get_SessionTicket(WOLFSSL* ssl, byte* buf, word32* bufSz);

Description:

This function copies the ticket member of the Session structure to the buffer.

Return Values:

SSL_SUCCESS - returned if the function executed without error.

BAD_FUNC_ARG - returned if one of the arguments was NULL or if the bufSz argument was 0.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - a byte pointer representing the memory buffer.

bufSz - a word32 pointer representing the buffer size.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz;  /*Initialize with buf size*/
…
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
      /*Nothing was written to the buffer*/
} else {
      /*the buffer holds the content from ssl->session.ticket */
}
```

See Also:
wolfSSL_UseSessionTicket
wolfSSL_set_SessionTicket

wolfSSL_lib_version_hex

Synopsis:
#include <wolfssl/ssl.h>

word32 wolfSSL_lib_version_hex(void);

Description:
This function returns the current library version in hexadecimal notation.

Return Values:
LILBWOLFSSL_VERSION_HEX - returns the hexidecimal version defined in wolfssl/version.h.

Parameters:

This function does not take any parameters.

Example:

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
	/*How to handle an unexpected value*/
} else {
	/*The expected result for libV */
}
```

See Also:

wolfSSL_lib_version

wolfSSL_SNI_Status

Synopsis:

#include <wolfssl/ssl.h>

byte wolfSSL_SNI_Status(WOLFSSL* ssl, byte type);

Description:

This function gets the status of an SNI object.

Return Values:

This function returns the **byte** value of the SNI struct's status member if the SNI is not NULL.

0 - if the SNI object is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

type - the SNI type.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
```

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#define AssertIntEQ(x, y) AssertInt(x, y, ==, !=)
…
Byte type = WOLFSSL_SNI_HOST_NAME;
char* request = (char*)&type;
AssertIntEQ(WOLFSSL_SNI_NO_MATCH, wolfSSL_SNI_Status(ssl, type));
…
```

See Also:
TLSX_SNI_Status
TLSX_SNI_find
TLSX_Find

wolfSSL_get_alert_history

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_alert_history(WOLFSSL* ssl, WOLFSSL_ALERT_HISTORY *h);

Description:
This function gets the alert history.

Return Values:
SSL_SUCCESS - returned when the function completed successfully. Either there was alert history or there wasn't, either way, the return value is SSL_SUCCESS.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

h - a pointer to a WOLFSSL_ALERT_HISTORY structure that will hold the WOLFSSL struct's **alert_history** member's value.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_ALERT_HISTORY* h;
...
```

```
wolfSSL_get_alert_history(ssl, h);
/*  h now has a copy of the ssl->alert_history  contents  */
```

See Also:
wolfSSL_get_error

wolfSSL_lib_version

Synopsis:
#include <wolfssl/ssl.h>

const char* wolfSSL_KeepArrays(void);

Description:
This function returns the current library version.

Return Values:
LIBWOLFSSL_VERSION_STRING - a const char pointer defining the version.

Parameters:

This function takes no parameters.

Example:

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
      /*Handle the mismatch case*/
}
```

See Also:
word32_wolfSSL_lib_version_hex

wolfSSL_CTX_UseCavium

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseCavium(WOLFSSL_CTX* ctx, int devId)

Description:

Forces provided **WOLFSSL_CTX** to use cavium.

Return Values:

SSL_SUCCESS: Successfully set cavium.

BAD_FUNC_ARG: Returns if *ctx* is null.

Parameters:

ctx - Pointer to **WOLFSSL_CTX** to use.

devId - The value to set the **ctx->devId** to.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseCavium(ctx, CAVIUM_DEV_ID) != SSL_SUCCESS)
{
    /* Error setting session ticket */
}
```

See Also:

wolfSSL_UseCavium

wolfSSL_UseCavium

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseCavium(WOLFSSL* ssl, int devId)

Description:

Forces provided **WOLFSSL** structure to use cavium.

Return Values:

SSL_SUCCESS: Success

BAD_FUNC_ARG: Returned if **ssl** is null.

Parameters:

ssl - Pointer to the **WOLFSSL** session. Created with wolfSSL_new()

devId - Value to set **ssl->devId** to.

Example:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = /* Some wolfSSL method */
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseCavium(ssl, CAVIUM_DEV_ID) != SSL_SUCCESS)
{
    /* Error setting session ticket */
}
```

See Also:

wolfSSL_CTX_UseCavium

wolfSSL_set_jobject

Synopsis:

#include <wolfssl/ssl.h>

void wolfSSL_set_jobject(WOLFSSL* ssl, void* objPtr);

Description:
This function sets the **jObjectRef** member of the WOLFSSL structure.

Return Values:
SSL_SUCCESS - returned if jObjectRef is properly set to objPtr.

SSL_FAILURE - returned if the function did not properly execute and jObjectRef is not set.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

objPtr - a void pointer that will be set to **jObjectRef.**

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = WOLFSSL_new();
void* objPtr = &obj;
...
if(wolfSSL_set_jobject(ssl, objPtr)){
      /*The success case*/
 }
```

See Also:
wolfSSL_get_jobject

wolfSSL_get_jobject

Synopsis:
#include <wolfssl/ssl.h>

void* wolfSSL_get_jobject(WOLFSSL* ssl);

Description:

This function returns the **jObjectRef** member of the WOLFSSL structure.

Return Values:

If the WOLFSSL struct is not NULL, the function returns the **jObjectRef** value.

NULL - returned if the WOLFSSL struct is NULL.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
      /*Success case*/
}
```

See Also:

wolfSSL_set_jobject

wolfSSL_BIO_ctrl_pending

Synopsis:

#include <wolfssl/ssl.h>

size_t wolfSSL_BIO_ctrl_pending(WOLFSSL_BIO* bio);

Description:

Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.

Return Values:

0 or greater: number of pending bytes.

Parameters:

bio - pointer to the WOLFSSL_BIO structure that has already been created

Example:

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();

….

pending = wolfSSL_BIO_ctrl_pending(bio);
```

See Also:

wolfSSL_BIO_make_bio_pair, wolfSSL_BIO_new

wolfSSL_BIO_get_mem_ptr

Synopsis:

#include <wolfssl/ssl.h>

BIO_get_mem_ptr ->
long wolfSSL_BIO_get_mem_ptr(WOLFSSL_BIO* bio, WOLFSSL_BUF_MEM** ptr);

Description:

This is a getter function for WOLFSSL_BIO memory pointer.

Return Values:

SSL_SUCCESS: On successfully getting the pointer SSL_SUCCESS is returned (currently value of 1).

SSL_FAILURE: Returned if NULL arguments are passed in (currently value of 0).

Parameters:

bio - pointer to the WOLFSSL_BIO structure for getting memory pointer.

ptr - structure that is currently a char*. Is set to point to bio's memory.

Example:

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;

// setup bio

wolfSSL_BIO_get_mem_ptr(bio, &pt);

//use pt
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem

wolfSSL_BIO_reset

Synopsis:

#include <wolfssl/ssl.h>

BIO_reset ->
int wolfSSL_BIO_reset(WOLFSSL_BIO* bio);

Description:

Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.

Return Values:

0: On successfully resetting the bio.

-1 (WOLFSSL_BIO_ERROR): Returned on bad input or unsuccessful reset.

Parameters:

bio - WOLFSSL_BIO structure to reset.

Example:

```
WOLFSSL_BIO* bio;
// setup bio
```

```
wolfSSL_BIO_reset(bio);

//use pt
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_ERR_load_BIO_strings

Synopsis:

#include <wolfssl/ssl.h>

ERR_load_BIO_strings ->
void wolfSSL_ERR_load_BIO_strings(void)

Description:

Do nothing. wolfSSL error string is statically defined.

Return Values:

None

Parameters:

none

Example:

```
    wolfSSL_ERR_load_BIO_strings();
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_s_socket

Synopsis:

#include <wolfssl/ssl.h>

BIO_s_socket ->
WOLFSSL_BIO_METHOD* wolfSSL_BIO_s_socket(void);

Description:

This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.

Return Values:

WOLFSSL_BIO_METHOD*: pointer to a WOLFSSL_BIO_METHOD structure that is a socket type

Parameters:

None

Example:

```
WOLFSSL_BIO* bio;

bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem

wolfSSL_BIO_set_fd

Synopsis:

#include <wolfssl/ssl.h>

BIO_set_fd ->
long wolfSSL_BIO_set_fd(WOLFSSL_BIO* bio, int fd, int closeF);

Description:

Sets the file descriptor for bio to use.

Return Values:

Returns SSL_SUCCESS (1).

Parameters:

bio - WOLFSSL_BIO structure to set fd.

fd - file descriptor to use.

closeF - flag for behavior when closing fd.

Example:

```
WOLFSSL_BIO* bio;
int fd;
// setup bio

wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_set_write_buf_size

Synopsis:

#include <wolfssl/ssl.h>

BIO_set_write_buf_size ->
int wolfSSL_BIO_set_write_buf_size(WOLFSSL_BIO* bio, long size;

Description:

This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

Return Values:

SSL_SUCCESS: On successfully setting the write buffer.
SSL_FAILURE: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to set fd.

size - size of buffer to allocate.

Example:

```
WOLFSSL_BIO* bio;
int ret;

bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// check return value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_make_bio_pair

Synopsis:

#include <wolfssl/ssl.h>

BIO_make_bio_pair ->
int wolfSSL_BIO_make_bio_pair(WOLFSSL_BIO* b1, WOLFSSL_BIO* b2);

Description:

This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.

Return Values:

SSL_SUCCESS: On successfully pairing the two bios.
SSL_FAILURE: If an error case was encountered.

Parameters:

b1 - WOLFSSL_BIO structure to set pair.

b2 - second WOLFSSL_BIO structure to complete pair.

Example:

```
WOLFSSL_BIO* bio;
WOLFSSL_BIO* bio2;
int ret;

bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
bio2 = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
ret = wolfSSL_BIO_make_bio_pair(bio, bio2);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_ctrl_reset_read_request

Synopsis:

#include <wolfssl/ssl.h>

BIO_ctrl_reset_read_request ->
int wolfSSL_BIO_ctrl_reset_read_request(WOLFSSL_BIO* bio);

Description:

This is used to set the read request flag back to 0.

Return Values:

SSL_SUCCESS: On successfully setting value.

SSL_FAILURE: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to set read request flag.

Example:

```
WOLFSSL_BIO* bio;
int ret;

...

ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem

wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_nread

Synopsis:

#include <wolfssl/ssl.h>

BIO_nread ->
int wolfSSL_BIO_nread(WOLFSSL_BIO* bio, char** buf, int num);

Description:

This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.

Return Values:

0 or greater: on success return the number of bytes to read

-1: on error case with nothing to read return -1 (WOLFSSL_BIO_ERROR)

Parameters:

bio - WOLFSSL_BIO structure to read from.

buf - pointer to set at beginning of read array.

num -number of bytes to try and read.

Example:

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// set up bio
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // try to read 10 bytes
// handle negative ret check
// read ret bytes from bufPt
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_nwrite

wolfSSL_BIO_nread0

Synopsis:

#include <wolfssl/ssl.h>

BIO_nread ->
int wolfSSL_BIO_nread0(WOLFSSL_BIO* bio, char** buf);

Description:

This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.

Return Values:

Greater than 0: on success return the number of bytes to read

Parameters:

bio - WOLFSSL_BIO structure to read from.

buf - pointer to set at beginning of read array.

Example:

```
WOLFSSL_BIO* bio;

char* bufPt;

int ret;

// set up bio

ret = wolfSSL_BIO_nread0(bio, &bufPt); // read as many bytes as possible
// handle negative ret check
// read ret bytes from bufPt
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_nwrite0

wolfSSL_BIO_nwrite

Synopsis:

#include <wolfssl/ssl.h>

BIO_nwrite ->
int wolfSSL_BIO_nwrite(WOLFSSL_BIO* bio, char** buf, int num);

Description:

Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.

Return Values:

Returns the number of bytes that can be written to the buffer pointer returned.

WOLFSSL_BIO_UNSET: -2 in the case that is not part of a bio pair

WOLFSSL_BIO_ERROR: -1 in the case that there is no more room to write to

Parameters:

bio - WOLFSSL_BIO structure to write to.

buf - pointer to buffer to write to.

num - number of bytes desired to be written.

Example:

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// set up bio
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // try to write 10 bytes
// handle negative ret check
// write ret bytes to bufPt
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_free, wolfSSL_BIO_nread

wolfSSL_BIO_puts

Synopsis:

#include <wolfssl/ssl.h>

BIO_puts ->
int wolfSSL_BIO_puts(WOLFSSL_BIO* bio, const char* data)

Description:

BIO_puts() tries to write a NUL-terminated string data to BIO bio.

Return Values:

Return the number of bytes that is successfully written.

SSL_FAILURE: o data was successfully written.

Parameters:

bio - WOLFSSL_BIO structure to write to.

data - pointer to buffer to write to.

Example:

```
WOLFSSL_BIO* bio;
char* data;

int ret;

// set up bio

ret = wolfSSL_BIO_puts (bio, &data, 10);

// handle negative ret check
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_free, wolfSSL_BIO_read

wolfSSL_BIO_set_fp

Synopsis:

#include <wolfssl/ssl.h>

BIO_set_fp ->
long wolfSSL_BIO_set_fp(WOLFSSL_BIO* bio, XFILE fp, int c);

Description:

This is used to set the internal file pointer for a BIO.

Return Values:

SSL_SUCCESS: On successfully setting file pointer.

SSL_FAILURE: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to set pair.

fp - file pointer to set in bio.

c - close file behavior flag.

Example:

```
WOLFSSL_BIO* bio;

XFILE fp;

int ret;

bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, fp, BIO_CLOSE);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem, wolfSSL_BIO_get_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_get_fp

Synopsis:

#include <wolfssl/ssl.h>

BIO_get_fp ->
long wolfSSL_BIO_get_fp(WOLFSSL_BIO* bio, XFILE fp);

Description:

This is used to get the internal file pointer for a BIO.

Return Values:

SSL_SUCCESS: On successfully getting file pointer.

SSL_FAILURE: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to set pair.

fp - file pointer to set in bio.

Example:

```
WOLFSSL_BIO* bio;

XFILE fp;

int ret;

bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_get_fp(bio, &fp);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem, wolfSSL_BIO_set_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_seek

Synopsis:

#include <wolfssl/ssl.h>

BIO_seek ->
int wolfSSL_BIO_seek(WOLFSSL_BIO* bio, int ofs);

Description:

This function adjusts the file pointer to the offset given. This is the offset from the head of the file.

Return Values:

0: On successfully seeking.

-1: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to set.

ofs - offset into file.

Example:

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, &fp);
// check ret value
ret  = wolfSSL_BIO_seek(bio, 3);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem, wolfSSL_BIO_set_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_write_filename

Synopsis:

#include <wolfssl/ssl.h>

BIO_write_filename ->
int wolfSSL_BIO_write_filename(WOLFSSL_BIO* bio, char* name);

Description:

This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

Return Values:

SSL_SUCCESS: On successfully opening and setting file.

SSL_FAILURE: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to set file.

name - name of file to write to.

Example:

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_write_filename(bio, “test.txt”);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_file, wolfSSL_BIO_set_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_get_mem_data

Synopsis:

#include <wolfssl/ssl.h>

BIO_get_mem_data ->
int wolfSSL_BIO_get_mem_data(WOLFSSL_BIO* bio, const byte** p);

Description:

This is used to set a byte pointer to the start of the internal memory buffer.

Return Values:

On success the size of the buffer is returned

SSL_FATAL_ERROR: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to get memory buffer of.

p - byte pointer to set to memory buffer.

Example:

```
WOLFSSL_BIO* bio;
const byte* p;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
```

```
ret  = wolfSSL_BIO_get_mem_data(bio, &p);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem, wolfSSL_BIO_set_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_get_mem_ptr

Synopsis:

#include <wolfssl/ssl.h>

BIO_get_mem_ptr ->
long wolfSSL_BIO_get_mem_ptr(WOLFSSL_BIO* bio, WOLFSSL_BUF_MEM** ptr);

Description:

This is used to get the internal memory pointer from a BIO.

Return Values:

SSL_SUCCESS: On successfully getting memory pointer.

SSL_FAILURE: If an error case was encountered.

Parameters:

bio - WOLFSSL_BIO structure to get memory pointer from.

ptr - pointer to WOLFSSL_BUF_MEM structure.

Example:

```
WOLFSSL_BIO* bio;

WOLFSSL_BUF_MEM* p;

int ret;

bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_get_mem_ptr(bio, &p);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem, wolfSSL_BIO_set_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

wolfSSL_BIO_set_mem_eof_return

Synopsis:

#include <wolfssl/ssl.h>

BIO_set_mem_eof_return ->
long wolfSSL_BIO_set_mem_eof_return(WOLFSSL_BIO* bio, int v);

Description:

This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.

Return Values:

Returns 0

Parameters:

bio - WOLFSSL_BIO structure to set end of file value.

v - value to set in bio.

Example:

```
WOLFSSL_BIO* bio;

int ret;

bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// check ret value
```

See Also:

wolfSSL_BIO_new, wolfSSL_BIO_s_mem, wolfSSL_BIO_set_fp
wolfSSL_BIO_new, wolfSSL_BIO_free

17.9 DTLS Specific

The functions in this section are specific to using DTLS with wolfSSL.

wolfSSL_dtls

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_dtls(WOLFSSL* ssl);

Description:
This function is used to determine if the SSL session has been configured to use DTLS.

Return Values:
If the SSL session (**ssl**) has been configured to use DTLS, this function will return 1, otherwise 0.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_dtls(ssl);
if (ret) {
      // SSL session has been configured to use DTLS
}
```

See Also:
wolfSSL_dtls_get_current_timeout
wolfSSL_dtls_get_peer
wolfSSL_dtls_got_timeout
wolfSSL_dtls_set_peer

wolfSSL_dtls_get_current_timeout

Synopsis:
#include <wolfssl/ssl.h>

wolfSSL_dtls_get_current_timeout(WOLFSSL* ssl);

Description:
This function returns the current timeout value in seconds for the WOLFSSL object. When using non-blocking sockets, something in the user code needs to decide when to check for available recv data and how long it has been waiting. The value returned by this function indicates how long the application should wait.

Return Values:
The current DTLS timeout value in seconds, or **NOT_COMPILED_IN** if wolfSSL was not built with DTLS support.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
int timeout = 0;
WOLFSSL* ssl;
...

timeout = wolfSSL_get_dtls_current_timeout(ssl);
printf("DTLS timeout (sec) = %d\n", timeout);
```

See Also:
wolfSSL_dtls
wolfSSL_dtls_get_peer
wolfSSL_dtls_got_timeout
wolfSSL_dtls_set_peer

wolfSSL_dtls_get_peer

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_dtls_get_peer(WOLFSSL* ssl, void* peer, unsigned int* peerSz);

Description:

This function gets the sockaddr_in (of size **peerSz**) of the current DTLS peer. The function will compare peerSz to the actual DTLS peer size stored in the SSL session. If the peer will fit into **peer**, the peer's sockaddr_in will be copied into **peer**, with peerSz set to the size of **peer**.

Return Values:

SSL_SUCCESS will be returned upon success.

SSL_FAILURE will be returned upon failure.

SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

peer - pointer to memory location to store peer's sockaddr_in structure.

peerSz - input/output size. As input, the size of the allocated memory pointed to by **peer**. As output, the size of the actual sockaddr_in structure pointed to by **peer**.

Example:

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...

ret = wolfSSL_dtls_get_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
      // failed to get DTLS peer
}
```

See Also:

wolfSSL_dtls_get_current_timeout
wolfSSL_dtls_got_timeout
wolfSSL_dtls_set_peer

wolfSSL_dtls

wolfSSL_dtls_got_timeout

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_dtls_got_timeout(WOLFSSL* ssl);

Description:
When using non-blocking sockets with DTLS, this function should be called on the WOLFSSL object when the controlling code thinks the transmission has timed out. It performs the actions needed to retry the last transmit, including adjusting the timeout value. If it has been too long, this will return a failure.

Return Values:

SSL_SUCCESS will be returned upon success

SSL_FATAL_ERROR will be returned if there have been too many retransmissions/ timeouts without getting a response from the peer.

NOT_COMPILED_IN will be returned if wolfSSL was not compiled with DTLS support.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

See the following files for usage examples:
<wolfssl_root>/examples/client/client.c
<wolfssl_root>/examples/server/server.c

See Also:
wolfSSL_dtls_get_current_timeout
wolfSSL_dtls_get_peer
wolfSSL_dtls_set_peer
wolfSSL_dtls

wolfSSL_dtls_set_peer

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_dtls_set_peer(WOLFSSL* ssl, void* peer, unsigned int peerSz);

Description:

This function sets the DTLS peer, **peer** (sockaddr_in) with size of **peerSz**.

Return Values:

SSL_SUCCESS will be returned upon success.

SSL_FAILURE will be returned upon failure.

SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

peer - pointer to peer's sockaddr_in structure.

peerSz - size of the sockaddr_in structure pointed to by **peer**.

Example:

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...

ret = wolfSSL_dtls_set_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
      // failed to set DTLS peer
}
```

See Also:

wolfSSL_dtls_get_current_timeout
wolfSSL_dtls_get_peer
wolfSSL_dtls_got_timeout
wolfSSL_dtls

wolfSSL_dtls_set_timeout_max

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_dtls_set_timeout_max(WOLFSSL* ssl, int timeout);

Description:
This function sets the maximum dtls timeout.

Return Values:
SSL_SUCCESS - returned if the function executed without an error.

BAD_FUNC_ARG - returned if the WOLFSSL struct is NULL or if the **timeout** argument is not greater than zero or is less than the **dtls_timeout_init** member of the WOLFSSL structure.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

timeout - an int type representing the dtls maximum timeout.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUTVAL;
...
int ret = wolfSSL_dtls_set_timeout_max(ssl);

if(!ret){
      /*Failed to set the max timeout*/
}
```

See Also:

wolfSSL_dtls_set_timeout_init
wolfSSL_dtls_got_timeout

wolfSSL_DTLS_SetCookieSecret

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_DTLS_SetCookieSecret(WOLFSSL* ssl, const byte* secret,
word32 secretSz);

Description:
This function sets a new dtls cookie secret.

Return Values:
0 - returned if the function executed without an error.

BAD_FUNC_ARG - returned if there was an argument passed to the function with an unacceptable value.

COOKIE_SECRET_SZ - returned if the secret size is 0.

MEMORY_ERROR - returned if there was a problem allocating memory for a new cookie secret.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

secret - a constant byte pointer representing the secret buffer.

secretSz - the size of the buffer.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
const* byte secret;
word32 secretSz; /*size of secret*/
…
```

```
if(!wolfSSL_DTLS_SetCookieSecret(ssl, secret, secretSz)){
      /*Code block for failure to set DTLS cookie secret*/
} else {
      /*Success! Cookie secret is set. */
}
```

See Also:

ForceZero
wc_RNG_GenerateBlock
XMEMCPY

wolfDTLSv1_2_client_method

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* wolfDTLSv1_2_client_method(void);

Description:

This function initializes the DTLS v1.2 client method.

Return Values:

This function returns a pointer to a new **WOLFSSL_METHOD** structure.

Parameters:

This function has no parameters.

Example:

```
wolfSSL_Init();

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_client_method());
…

WOLFSSL* ssl = wolfSSL_new(ctx);
…
```

See Also:

wolfSSL_Init
wolfSSL_CTX_new

wolfSSL_dtls_export

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_dtls_export(WOLFSSL* ssl, unsigned char* buf, unsigned int* sz);

Description:

The wolfSSL_dtls_export() function is used to serialize a WOLFSSL session into the provided buffer. Allows for less memory overhead than using a function callback for sending a session and choice over when the session is serialized. If buffer is NULL when passed to function then sz will be set to the size of buffer needed for serializing the WOLFSSL session.

Return Values:

If successful, the amount of the buffer used will be returned.

All unsuccessful return values will be less than 0.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - buffer to hold serialized session.

sz - size of buffer.

Example:

```
WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
```

```
bufSz = MAX;

...

ret = wolfSSL_dtls_export(ssl, buf, bufSz);
if (ret < 0) {
// handle error case
}
...
```

See Also:

wolfSSL_new

wolfSSL_CTX_new

wolfSSL_CTX_dtls_set_export

wolfSSL_dtls_import

wolfSSL_dtls_import

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_dtls_import(WOLFSSL* ssl, unsigned char* buf, unsigned int sz);

Description:

The wolfSSL_dtls_import() function is used to parse in a serialized session state. This allows for picking up the connection after the handshake has been completed.

Return Values:

If successful, the amount of the buffer read will be returned.

All unsuccessful return values will be less than 0.

If a version mismatch is found ie DTLS v1 and ctx was set up for DTLS v1.2 then VERSSION_ERROR is returned.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

buf - serialized session to import.

sz - size of serialized session buffer.

Example:

```
WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;

...
//get information sent from wc_dtls_export function and place it in buf
fread(buf, 1, bufSz, input);

ret = wolfSSL_dtls_import(ssl, buf, bufSz);
if (ret < 0) {
// handle error case
}

// no wolfSSL_accept needed since handshake was already done

...
ret = wolfSSL_write(ssl) and wolfSSL_read(ssl);
...
```

See Also:
wolfSSL_new
wolfSSL_CTX_new
wolfSSL_CTX_dtls_set_export

wolfSSL_dtls_set_export

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_dtls_set_export(WOLFSSL* ssl, wc_dtls_export func);

Description:

The wolfSSL_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed.

Return Values:

If successful, the call will return **SSL_SUCCESS**.

If null or not expected arguments are passed in **BAD_FUNC_ARG** is returned.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

func - wc_dtls_export function to use when exporting a session.

Example:

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);

// body of send session (wc_dtls_export) that passses buf (serialized
session) to destination

WOLFSSL* ssl;
int ret;

...

ret = wolfSSL_dtls_set_export(ssl, send_session);
if (ret != SSL_SUCCESS) {
// handle error case
}

...
ret = wolfSSL_accept(ssl);
...
```

See Also:

wolfSSL_new

wolfSSL_CTX_new

wolfSSL_CTX_dtls_set_export

wolfSSL_CTX_dtls_set_export

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_dtls_set_export(WOLFSSL_CTX* ctx, wc_dtls_export func);

Description:

The wolfSSL_CTX_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed.

Return Values:

If successful, the call will return **SSL_SUCCESS**.

If null or not expected arguments are passed in **BAD_FUNC_ARG** is returned.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

func - wc_dtls_export function to use when exporting a session.

Example:

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);

// body of send session (wc_dtls_export) that passses buf (serialized
session) to destination
```

```
WOLFSSL_CTX* ctx;
int ret;

...

ret = wolfSSL_CTX_dtls_set_export(ctx, send_session);
if (ret != SSL_SUCCESS) {
// handle error case
}

...
ret = wolfSSL_accept(ssl);
...
```

See Also:

wolfSSL_new
wolfSSL_CTX_new
wolfSSL_dtls_set_export
Static buffer use

wolfSSL_CTX_load_static_memory

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_load_static_memory(WOLFSSL_CTX** ctx, wolfSSL_method_func method, unsigned char* buf, unsigned int sz, int flag, int max);

Description:

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);.
Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place.

The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.

0 - default general memory
WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO.
WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime.
WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Return Values:

If successful, **SSL_SUCCESS** will be returned.

All unsuccessful return values will be less than 0 or equal to **SSL_FAILURE**.

Parameters:

ctx - address of pointer to a WOLFSSL_CTX structure.

method - function to create protocol. (should be NULL if ctx is not also NULL)

buf - memory to use for all operations.

sz - size of memory buffer being passed in.

flag - type of memory.

max - max concurrent operations.

Example:

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
```

```
unsigned char IO[MAX];
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;

...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
memory, memorySz, 0, MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
      // handle error case
}


// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
      // handle error case
}
...
```

See Also:

wolfSSL_CTX_new
wolfSSL_CTX_is_static_memory
wolfSSL_is_static_memory

wolfSSL_CTX_is_static_memory

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx, WOLFSSL_MEM_STATS* mem_stats);

Description:

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Return Values:

A value of **1** is returned if using static memory for the CTX is true.

0 is returned if not using static memory.

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

mem_stats - structure to hold information about static memory usage.

Example:

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;

...
//get information about static memory with CTX
ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);
if (ret == 1) {
// handle case of is using static memory
 // print out or inspect elements of mem_stats
}

if (ret == 0) {
//handle case of ctx not using static memory
}
…
```

See Also:

wolfSSL_CTX_new
wolfSSL_CTX_load_static_memory
wolfSSL_is_static_memory

wolfSSL_is_static_memory

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_is_static_memory(WOLFSSL* ssl, WOLFSSL_MEM_CONN_STATS* mem_stats);

Description:

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Return Values:

A value of **1** is returned if using static memory for the CTX is true.

0 is returned if not using static memory.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

mem_stats - structure to contain static memory usage.

Example:

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);
if (ret == 1) {
// handle case when is static memory
```

```
// investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
}

...
```

See Also:

wolfSSL_new

wolfSSL_CTX_is_static_memory

wolfDTLSv1_2_server_method

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* wolfDTLSv1_2_server_method(void);

Description:

This function creates and initializes a WOLFSSL_METHOD for the server side.

Return Values:

This function returns a WOLFSSL_METHOD pointer.

Parameters:

This function takes no parameters.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_server_method());
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
```

See Also:

wolfSSL_CTX_new

17.10 Memory Abstraction Layer

The functions in this section are used when an application sets its own memory handling functions by using the wolfSSL memory abstraction layer.

wolfSSL_Malloc

Synopsis:
#include <wolfssl/wolfcrypt/memory.h>

void* wolfSSL_Malloc(size_t size)

Description:
This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators().

Return Values:
If successful, this function returns a pointer to allocated memory. If there is an error, NULL will be returned. Specific return values may be dependent on the underlying memory allocation function being used (if not using the default malloc()).

Parameters:

size - number of bytes to allocate.

Example:

```
char* buffer;

buffer = (char*) wolfSSL_Malloc(20);
if (buffer == NULL) {
      // failed to allocate memory
}
```

See Also:
wolfSSL_Free
wolfSSL_Realloc

wolfSSL_SetAllocators

wolfSSL_Realloc

Synopsis:
#include <wolfssl/wolfcrypt/memory.h>

void* wolfSSL_Realloc(void *ptr, size_t size)

Description:
This function is similar to realloc(), but calls the memory re-allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators().

Return Values:
If successful, this function returns a pointer to re-allocated memory. This may be the same pointer as **ptr**, or a new pointer location. If there is an error, NULL will be returned. Specific return values may be dependent on the underlying memory re-allocation function being used (if not using the default realloc()).

Parameters:

ptr - pointer to the previously-allocated memory, to be reallocated.

size - number of bytes to allocate.

Example:

```
char* buffer;

buffer = (char*) wolfSSL_Realloc(30);
if (buffer == NULL) {
      // failed to re-allocate memory
}
```

See Also:
wolfSSL_Free
wolfSSL_Malloc
wolfSSL_SetAllocators

wolfSSL_Free

Synopsis:
#include <wolfssl/wolfcrypt/memory.h>

void wolfSSL_Free(void* ptr)

Description:
This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators().

Return Values:

This function does not have a return value.

Parameters:

ptr - pointer to the memory to be freed.

Example:

```
char* buffer;
...

wolfSSL_Free(buffer);
```

See Also:
wolfSSL_Alloc
wolfSSL_Realloc
wolfSSL_SetAllocators

wolfSSL_SetAllocators

Synopsis:
#include <wolfssl/wolfcrypt/memory.h>

int wolfSSL_SetAllocators(wolfSSL_Malloc_cb malloc_function,
wolfSSL_Free_cb free_function,
wolfSSL_Realloc_cb realloc_function);

typedef void *(*wolfSSL_Malloc_cb)(size_t size);
typedef void (*wolfSSL_Free_cb)(void *ptr);
typedef void *(*wolfSSL_Realloc_cb)(void *ptr, size_t size);

Description:

This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.

Return Values:

If successful this function will return 0.

BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Parameters:

malloc_function - memory allocation function for wolfSSL to use. Function signature must match wolfSSL_Malloc_cb prototype, above.

free_function - memory free function for wolfSSL to use. Function signature must match wolfSSL_Free_cb prototype, above.

realloc_function - memory re-allocation function for wolfSSL to use. Function signature must match wolfSSL_Realloc_cb prototype, above.

Example:

```
int ret = 0;

// Memory function prototypes
void* MyMalloc(size_t size);
void  MyFree(void* ptr);
void* MyRealloc(void* ptr, size_t size);

// Register custom memory functions with wolfSSL
ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
      // failed to set memory functions
}

void* MyMalloc(size_t size)
```

```
{
      // custom malloc function
}

void MyFree(void* ptr)
{
      // custom free function
}

void* MyRealloc(void* ptr, size_t size)
{
      // custom realloc function
}
```

See Also:
NA

17.11 Certificate Manager

The functions in this section are part of the wolfSSL Certificate Manager. The Certificate Manager allows applications to load and verify certificates external to the SSL/TLS connection.

wolfSSL_CertManagerDisableCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerDisableCRL(WOLFSSL_CERT_MANGER* cm);

Description:
Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

Example:

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;

...

ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
      // error disabling cert manager
}

...
```

See Also:
wolfSSL_CertManagerEnableCRL

wolfSSL_CertManagerEnableCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerEnableCRL(WOLFSSL_CERT_MANGER* cm, int options);

Description:
Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.

Return Values:
If successful the call will return **SSL_SUCCESS**.

NOT_COMPILED_IN will be returned if wolfSSL was not built with CRL enabled.

MEMORY_E will be returned if an out of memory condition occurs.

BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

SSL_FAILURE will be returned if the CRL context cannot be initialized properly.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

options - options to use when enabling the Certification Manager, **cm**.

Example:

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
      // error enabling cert manager
}

...
```

See Also:
wolfSSL_CertManagerDisableCRL

wolfSSL_CertManagerFree

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CertManagerFree(WOLFSSL_CERT_MANGER* cm);

Description:
Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.

Return Values:
No return value is used.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

Example:

```
WOLFSSL_CERT_MANAGER* cm;
...

wolfSSL_CertManagerFree(cm);
```

See Also:

wolfSSL_CertManagerNew

wolfSSL_CertManagerLoadCA

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CertManagerLoadCA(WOLFSSL_CERT_MANGER* cm,
const char* file, const char* path);

Description:

Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.

Return Values:

If successful the call will return **SSL_SUCCESS**.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

SSL_FATAL_ERROR - will be returned upon failure.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

file - pointer to the name of the file containing CA certificates to load.

path - pointer to the name of a directory path containing CA certificates to load. The NULL pointer may be used if no certificate directory is desired.

Example:

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerLoadCA(cm, "path/to/cert-file.pem", 0);
if (ret != SSL_SUCCESS) {
      // error loading CA certs into cert manager
}
```

See Also:
wolfSSL_CertManagerVerify

wolfSSL_CertManagerNew

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* wolfSSL_CertManagerNew(void);

Description:
Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Return Values:
If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.

NULL will be returned for an error state.

Parameters:

There are no parameters for this function.

Example:

```
WOLFSSL_CERT_MANAGER* cm;

cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
      // error creating new cert manager
}
```

See Also:
wolfSSL_CertManagerFree

wolfSSL_CertManagerVerify

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerVerify(WOLFSSL_CERT_MANGER* cm, const char* fname,
int fomat);

Description:
Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Return Values:
If successful the call will return **SSL_SUCCESS**.

ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.

ASN_SIG_OID_E will be returned if the signature type is not supported.

CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.

CRL_MISSING is an error that is returned if a current issuer CRL is not available.

ASN_BEFORE_DATE_E will be returned if the current date is before the before date.

ASN_AFTER_DATE_E will be returned if the current date is after the after date.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

fname - pointer to the name of the file containing the certificates to verify.

format - format of the certificate to verify - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, "path/to/cert-file.pem",
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      // error verifying certificate
}
```

See Also:
wolfSSL_CertManagerLoadCA
wolfSSL_CertManagerVerifyBuffer

wolfSSL_CertManagerVerifyBuffer

Synopsis:
#include <wolfssl/ssl.h>

```
int wolfSSL_CertManagerVerifyBuffer(WOLFSSL_CERT_MANGER* cm,
                                    const byte* buff, long sz, int format);
```

Description:
Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Return Values:
If successful the call will return **SSL_SUCCESS**.

ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.

ASN_SIG_OID_E will be returned if the signature type is not supported.

CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.

CRL_MISSING is an error that is returned if a current issuer CRL is not available.

ASN_BEFORE_DATE_E will be returned if the current date is before the before date.

ASN_AFTER_DATE_E will be returned if the current date is after the after date.

SSL_BAD_FILETYPE will be returned if the file is the wrong format.

SSL_BAD_FILE will be returned if the file doesn’t exist, can’t be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

ASN_INPUT_E will be returned if Base16 decoding fails on the file.

BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

buff - buffer containing the certificates to verify.

sz - size of the buffer, **buf**.

format - format of the certificate to verify, located in **buf** - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Example:

```
int ret = 0;
int sz = 0;
WOLFSSL_CERT_MANAGER* cm;
byte certBuff[...];
...

ret = wolfSSL_CertManagerVerifyBuffer(cm, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      // error verifying certificate
}
```

See Also:
wolfSSL_CertManagerLoadCA
wolfSSL_CertManagerVerify

wolfSSL_CertManagerCheckOCSP

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerCheckOCSP(WOLFSSL_CERT_MANAGER* cm,
byte* der, int sz);

Description:

The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.

Return Values:
SSL_SUCCESS - returned on successful execution of the function. The ocspEnabled member of the WOLFSSL_CERT_MANAGER is enabled.

BAD_FUNC_ARG - returned if the WOLFSSL_CERT_MANAGER structure is NULL or if an argument value that is not allowed is passed to a subroutine.

MEMORY_E - returned if there is an error allocating memory within this function or a subroutine.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

der - a byte pointer to the certificate.

sz - an int type representing the size of the DER cert.

Example:
```
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; /*size of der */
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:
ParseCertRelative
CheckCertOCSP

wolfSSL_CertManagerUnloadCAs

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerUnloadCAs(WOLFSSL_CERT_MANAGER* cm);

Description:
This function unloads the CA signer list.

Return Values:
SSL_SUCCESS - returned on successful execution of the function.

BAD_FUNC_ARG - returned if the WOLFSSL_CERT_MANAGER is NULL.

BAD_MUTEX_E - returned if there was a mutex error.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnloadCAs(ctx->cm) != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:
FreeSignerTable
UnlockMutex

wolfSSL_CertManagerSetOCSPOverrideURL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerSetOCSPOverrideURL(WOLFSSL_CERT_MANAGER* cm,
const char* url);

Description:
The function copies the url to the ocspOverrideURL member of the

WOLFSSL_CERT_MANAGER structure.

Return Values:
SSL_SUCCESS - the function was able to execute as expected.

BAD_FUNC_ARG - the WOLFSSL_CERT_MANAGER struct is NULL.

MEMEORY_E - Memory was not able to be allocated for the ocspOverrideURL member of the certificate manager.

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new().

Example:

```
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
const char* url;
…
int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url)
…
if(wolfSSL_CertManagerSetOCSPOverrideURL(ssl->ctx->cm, url) != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:
ocspOverrideURL
wolfSSL_SetOCSP_OverrideURL

wolfSSL_CertManagerLoadCRL

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerLoadCRL(WOLFSSL_CERT_MANAGER* cm,
                  const char* path, int type, int monitor);

Description:
Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking.

Return Values:

SSL_SUCCESS - if there is no error in wolfSSL_CertManagerLoadCRL and if LoadCRL returns successfully.

BAD_FUNC_ARG - if the WOLFSSL_CERT_MANAGER struct is NULL.

SSL_FATAL_ERROR - if wolfSSL_CertManagerEnableCRL returns anything other than SSL_SUCCESS.

BAD_PATH_ERROR - if the path is NULL.

MEMORY_E - if LoadCRL fails to allocate heap memory.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

path - a constant char pointer holding the CRL path.

type - type of certificate to be loaded.

monitor - requests monitoring in LoadCRL().

Example:

```
int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type,
                        int monitor);
…
wolfSSL_CertManagerLoadCRL(ssl->ctx->cm, path, type, monitor);
```

See Also:

wolfSSL_CertManagerEnableCRL
wolfSSL_LoadCRL

wolfSSL_CertManagerLoadCABuffer

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerLoadCABuffer(WOLFSSL_CERT_MANAGER* cm,
const unsigned char* in, long sz, int format);

Description:
Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.

Return Values:

SSL_FATAL_ERROR is returned if the WOLFSSL_CERT_MANAGER struct is NULL or if wolfSSL_CTX_new() returns NULL.

SSL_SUCCESS is returned for a successful execution.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

in - buffer for cert information.

sz - length of the buffer.

format - certificate format, either PEM or DER.

Example:

```
WOLFSSL_CERT_MANAGER* cm = (WOLFSSL_CERT_MANAGER*)vp;
…
const unsigned char* in;
long sz;
int format;
…
if(wolfSSL_CertManagerLoadCABuffer(vp, sz, format) != SSL_SUCCESS){
      /*Error returned. Failure case code block. */
}
```

See Also:
wolfSSL_CTX_load_verify_buffer
ProcessChainBuffer
ProcessBuffer
cm_pick_method

wolfSSL_CertManagerUnload_trust_peers

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerUnload_trust_peers(WOLFSSL_CERT_MANAGER* cm);

Description:
The function will free the Trusted Peer linked list and unlocks the trusted peer list.

Return Values:
SSL_SUCCESS if the function completed normally.

BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.

BAD_MUTEX_E mutex error if tpLock, a member of the WOLFSSL_CERT_MANAGER struct, is 0 (nill).

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

Example:

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(/*Protocol define*/);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnload_trust_peers(cm) != SSL_SUCCESS){
      /*The function did not execute successfully. */
}
```

See Also:

UnLockMutex

wolfSSL_CertManagerEnableOCSP

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CertManagerEnableOCSP(WOLFSSL_CERT_MANAGER* cm,
int options) ;

Description:
Turns on OCSP if it's turned off and if compiled with the set option available.

Return Values:
SSL_SUCCESS returned if the function call is successful.

BAD_FUNC_ARG if cm struct is NULL.

MEMORY_E if WOLFSSL_OCSP struct value is NULL.

SSL_FAILURE initialization of WOLFSSL_OCSP struct fails to initialize.

NOT_COMPILED_IN build not compiled with correct feature enabled.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

options - used to set values in WOLFSSL_CERT_MANAGER struct.

Example:

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(/*protocol method*/);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
int options;
```

```
…
if(wolfSSL_CertManagerEnableOCSP(ssl->ctx->cm, options) != SSL_SUCCESS){
      /*Failure case. */
}
```

See Also:
wolfSSL_CertManagerNew

wolfSSL_BN_div

Synopsis:
#include <wolfssl/ssl.h>
#include <wolfssl/openssl/bn.h>

int wolfSSL_BN_div(WOLFSSL_BIGNUM* r, WOLFSSL_BIGNUM* m,

const WOLFSSL_BIGNUM* a, const WOLFSSL_BIGNUM* b,

const WOLFSSL_BN_CTX* c)

Description:
This function divides a by b and returns the quotient in r and the remainder in m (r=a/b, m=a%b). Either r or m may be NULL, in such case the value is not returned respectively. The quotient is rounded towards zero.

Return Values:
SSL_SUCCESS returned if the function call is successful.

Parameters:
r - the quotient (a/b)

m - the remainder (a%b)

a- the dividend

b - the divisor

c - pointer to a WOLFSSL_BN_CTX structure

Example:

```
BIGNUM *r, *m, *a,*b;
BN_CTX *c;
unsigned long wa,wb;

a = BN_new();
b = BN_new();
r = BN_new();
m = BN_new();

wa = 100;
wb = 30;
BN_set_word(a, wa);
BN_set_word(b, wb);
c = NULL;

if(BN_div(r, m, a, b, c)!= SSL_SUCCESS){
      /*Failure case. */
};

BN_free(a);
BN_free(b);
```

wolfSSL_BN_mod_inverse

Synopsis:

#include <wolfssl/openssl/bn.h>

WOLFSSL_BIGNUM *wolfSSL_BN_mod_inverse(WOLFSSL_BIGNUM* r, WOLFSSL_BIGNUM* a, const WOLFSSL_BIGNUM* n, WOLFSSL_BN_CTX *ctx);

Description:

This function compute the inverse of a modulo n places the results in r ((a*r)%n == 1).If r is NULL, a new BIGNUM is created.

Return Values:

Returns a pointer to computed bignum value and **NULL** on failure.

Parameters:

r - placeholder for computed mod inverse bignum value
a - bignum argument to compute mod inverse in (a*r)%n == 1
n - bignum argument to compute mod inverse in (a*r)%n == 1
ctx - bignum context

Example:

```
unsigned char value[1];
WOLFSSL_BIGNUM* r,a,n,val;

value[0] = 0x02;
wolfSSL_BN_bin2bn(value, sizeof(value), a);
value[0] = 0x05;
wolfSSL_BN_bin2bn(value, sizeof(value), n);

r = wolfSSL_BN_new();
val = wolfSSL_mod_inverse(r,a,n);
printf("mod inverse = %x\n",wolfSSL_BN_bn2bin(r,value));

wolfSSL_BN_free(a);
wolfSSL_BN_free(n);
wolfSSL_BN_free(r);
```

17.12 OpenSSL Compatibility Layer

The functions in this section are part of wolfSSL's OpenSSL Compatibility Layer. These functions are only available when wolfSSL has been compiled with the OPENSSL_EXTRA define or the OPENSSL_ALL define.

wolfSSL_X509_get_serial_number

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509, byte* in,
int* inOutSz);

Description:

Retrieves the peer's certificate serial number. The serial number buffer (**in**) should be at least 32 bytes long and be provided as the ***inOutSz** argument as input. After calling the function ***inOutSz** will hold the actual length in bytes written to the **in** buffer.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG will be returned if a bad function argument was encountered.

See Also:
SSL_get_peer_certificate

wolfSSL_get_sessionID

Synopsis:
#include <wolfssl/ssl.h>

const unsigned char* wolfSSL_get_sessionID(const WOLFSSL_SESSION* session);

Description:
Retrieves the session's ID. The session ID is always 32 bytes long.

Return Values:
The session ID.

See Also:
SSL_get_session()

wolfSSL_get_peer_chain

Synopsis:
#include <wolfssl/ssl.h>

X509_CHAIN* wolfSSL_get_peer_chain(WOLFSSL* ssl);

Description:
Retrieves the peer's certificate chain.

Return Values:
If successful the call will return the peer's certificate chain.

0 will be returned if an invalid WOLFSSL pointer is passed to the function.

See Also:
wolfSSL_get_chain_count
wolfSSL_get_chain_length
wolfSSL_get_chain_cert
wolfSSL_get_chain_cert_pem

wolfSSL_get_chain_count

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_chain_count(WOLFSSL_X509_CHAIN* chain);

Description:
Retrieves the peer's certificate chain count.

Return Values:
If successful the call will return the peer's certificate chain count.

0 will be returned if an invalid chain pointer is passed to the function.

See Also:
wolfSSL_get_peer_chain
wolfSSL_get_chain_length
wolfSSL_get_chain_cert
wolfSSL_get_chain_cert_pem

wolfSSL_get_chain_length

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_get_chain_length(WOLFSSL_X509_CHAIN* chain, int idx);

Description:
Retrieves the peer's ASN1.DER certificate length in bytes at index (**idx**).

Return Values:
If successful the call will return the peer's certificate length in bytes by index.

0 will be returned if an invalid chain pointer is passed to the function.

See Also:
wolfSSL_get_peer_chain
wolfSSL_get_chain_count
wolfSSL_get_chain_cert
wolfSSL_get_chain_cert_pem

wolfSSL_get_chain_cert

Synopsis:
#include <wolfssl/ssl.h>

byte* wolfSSL_get_chain_cert(WOLFSSL_X509_CHAIN* chain, int idx);

Description:
Retrieves the peer's ASN1.DER certificate at index (**idx**).

Return Values:
If successful the call will return the peer's certificate by index.

0 will be returned if an invalid chain pointer is passed to the function.

See Also:
wolfSSL_get_peer_chain
wolfSSL_get_chain_count
wolfSSL_get_chain_length
wolfSSL_get_chain_cert_pem

wolfSSL_get_chain_cert_pem

Synopsis:
#include <wolfssl/ssl.h>

unsigned char* wolfSSL_get_chain_cert_pem(WOLFSSL_X509_CHAIN* chain, int idx);

Description:
Retrieves the peer's PEM certificate at index (**idx**).

Return Values:
If successful the call will return the peer's certificate by index.

0 will be returned if an invalid chain pointer is passed to the function.

See Also:
wolfSSL_get_peer_chain
wolfSSL_get_chain_count
wolfSSL_get_chain_length
wolfSSL_get_chain_cert

wolfSSL_PemCertToDer

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_PemCertToDer(const char* fileName, unsigned char* derBuf, int derSz);

Description:
Loads the PEM certificate from **fileName** and converts it into DER format, placing the result into **derBuffer** which is of size **derSz**.

Return Values:
If successful the call will return the number of bytes written to **derBuffer**.

SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.

MEMORY_E will be returned if an out of memory condition occurs.

SSL_NO_PEM_HEADER will be returned if the PEM certificate header can't be found.

BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Parameters:

filename - pointer to the name of the PEM-formatted certificate for conversion.

derBuffer - the buffer for which the converted PEM certificate will be placed in DER format.

derSz - size of derBuffer.

Example:

```
int derSz;
byte derBuf[...];

derSz = wolfSSL_PemCertToDer("./cert.pem", derBuf, sizeof(derBuf));
```

See Also:
SSL_get_peer_certificate

wolfSSL_CTX_use_RSAPrivateKey_file

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_use_RSAPrivateKey_file(WOLFSSL_CTX* ctx,const char* file,
int format);

Description:
This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (--enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_CTX_use_PrivateKey_file() function.

The **file** argument contains a pointer to the RSA private key file, in the format specified by **format**.

Return Values:
If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The input key file is in the wrong format, or the wrong format has been given using the "format" argument
- file doesn't exist, can't be read, or is corrupted
- an out of memory condition occurs

Parameters:

ctx - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()

file - a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL context, with format as specified by **format**.

format - the encoding type of the RSA private key specified by **file**. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx;

...

ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, "./server-key.pem",
                                       SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      // error loading private key file
}

...
```

See Also:
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_use_RSAPrivateKey_file
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_PrivateKey_file

wolfSSL_use_certificate_file

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_use_certificate_file(WOLFSSL* ssl, const char* file, int format);

Description:

This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the **file** argument. The **format** argument specifies the format type of the file - either **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The file is in the wrong format, or the wrong format has been given using the “format” argument
- file doesn’t exist, can’t be read, or is corrupted
- an out of memory condition occurs
- Base16 decoding fails on the file

Parameters:

ssl - a pointer to a WOLFSSL structure, created with wolfSSL_new().

file - a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL session, with format as specified by **format**.

format - the encoding type of the certificate specified by **file**. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

Example:

```
int ret = 0;
WOLFSSL* ssl;

...

ret = wolfSSL_use_certificate_file(ssl, "./client-cert.pem",
                                 SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      // error loading cert file
}

...
```

See Also:

wolfSSL_CTX_use_certificate_buffer
wolfSSL_CTX_use_certificate_file
wolfSSL_use_certificate_buffer

wolfSSL_use_PrivateKey_file

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_use_PrivateKey_file(WOLFSSL* ssl, const char* file, int format);

Description:

This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the **file** argument. The **format** argument specifies the format type of the file - **SSL_FILETYPE_ASN1** or **SSL_FILETYPE_PEM**.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The file is in the wrong format, or the wrong format has been given using the “format” argument
- The file doesn’t exist, can’t be read, or is corrupted
- An out of memory condition occurs
- Base16 decoding fails on the file
- The key file is encrypted but no password is provided

Parameters:

ssl - a pointer to a WOLFSSL structure, created with wolfSSL_new().

file - a pointer to the name of the file containing the key file to be loaded into the wolfSSL SSL session, with format as specified by **format**.

format - the encoding type of the key specified by **file**. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

Example:

```
int ret = 0;
WOLFSSL* ssl;

...

ret = wolfSSL_use_PrivateKey_file(ssl, "./server-key.pem",
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      // error loading key file
}

...
```

See Also:
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_use_PrivateKey_buffer

wolfSSL_use_certificate_chain_file

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_use_certificate_chain_file(WOLFSSL* ssl, const char* file);

Description:
This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the **file** argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

Return Values:
If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The file is in the wrong format, or the wrong format has been given using the "format" argument
- file doesn't exist, can't be read, or is corrupted
- an out of memory condition occurs

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new()

file - a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL session. Certificates must be in PEM format.

Example:

```
int ret = 0;
WOLFSSL* ctx;

...

ret = wolfSSL_use_certificate_chain_file(ssl, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
      // error loading cert file
}

...
```

See Also:

wolfSSL_CTX_use_certificate_chain_file
wolfSSL_CTX_use_certificate_chain_buffer
wolfSSL_use_certificate_chain_buffer

wolfSSL_use_RSAPrivateKey_file

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_use_RSAPrivateKey_file(WOLFSSL* ssl,const char* file, int format);

Description:

This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (--enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_use_PrivateKey_file() function.

The **file** argument contains a pointer to the RSA private key file, in the format specified by **format**.

Return Values:

If successful the call will return **SSL_SUCCESS**, otherwise **SSL_FAILURE** will be returned. If the function call fails, possible causes might include:

- The input key file is in the wrong format, or the wrong format has been given using the "format" argument
- file doesn't exist, can't be read, or is corrupted
- an out of memory condition occurs

Parameters:

ssl - a pointer to a WOLFSSL structure, created using wolfSSL_new()

file - a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL session, with format as specified by **format**.

format - the encoding type of the RSA private key specified by **file**. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

Example:

```
int ret = 0;
WOLFSSL* ssl;

...

ret = wolfSSL_use_RSAPrivateKey_file(ssl, "./server-key.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
      // error loading private key file
}

...
```

See Also:

wolfSSL_CTX_use_RSAPrivateKey_file
wolfSSL_CTX_use_PrivateKey_buffer
wolfSSL_CTX_use_PrivateKey_file
wolfSSL_use_PrivateKey_buffer
wolfSSL_use_PrivateKey_file

wolfSSL_PKCS5_PBKDF2_HMAC_SHA1

Synopsis:

#include <wolfssl/openssl/evp.h>

int wolfSSL_PKCS5_PBKDF2_HMAC_SHA1(const char *pass, int passlen, const unsigned char *salt, int saltlen, int iter, int keylen, unsigned char *out);

Description:

This function derives a key from a password using a salt and iteration count as specified in RFC2898.

Return Values:

Return **1** on success or **0** on error.

Parameters:

pass - password
passlen - password length
salt - salt
saltlen - salt length
iter - iteration count
keylen - key length

Example:

```
const char *pass = "pass";
const unsigned char *salt = (unsigned char *)"salt";
unsigned char *out = malloc(256);
int iter = 100;
int ret = 0;
int pass_len = 0;
int salt_len = 0;

pass_len = strlen(pass);
salt_len = strlen(salt);

ret =
WolfSSL_PBKDF2_HMAC_SHA1(passwd,pass_len,salt,salt_len,iter,SHA_DIGEST_SIZE,o
ut);
```

```
free(out);
```

See Also:

wolfSSL_PKCS12_parse

Synopsis:

#include <wolfssl/ssl.h>

PKCS12_parse ->
int wolfSSL_PKCS12_parse(WC_PKCS12* pkcs12, const char* paswd,
WOLFSSL_EVP_PKEY** pkey, WOLFSSL_X509** cert,
STACK_OF(WOLFSSL_X509)** stack);

Description:

PKCS12 can be enabled with adding –enable-opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (--enable-des3 –enable-arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files.

wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos.

This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates.

At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

Return Values:

SSL_SUCCESS: On successfully parsing PKCS12.

SSL_FAILURE: If an error case was encountered.

Parameters:

pkcs12 - WC_PKCS12 structure to parse.

paswd - password for decrypting PKCS12.

pkey - structure to hold private key decoded from PKCS12.

cert - structure to hold certificate decoded from PKCS12.

stack - optional stack of extra certificates.

Example:

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;

//bio loads in PKCS12 file

wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)

//use cert, pkey, and optionally certs stack
```

See Also:

wolfSSL_d2i_PKCS12_bio, wc_PKCS12_free

wolfSSL_d2i_PKCS12_bio

Synopsis:

#include <wolfssl/ssl.h>

d2i_PKCS12_bio ->
WC_PKCS12* wolfSSL_d2i_PKCS12_bio(WOLFSSL_BIO* bio, WC_PKCS12** pkcs12);

Description:

wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling

Return Values:

WC_PKCS12*: pointer to a WC_PKCS12 structure. If function failed it will return NULL.

Parameters:

bio - WOLFSSL_BIO structure to read PKCS12 buffer from.
pkcs12 - WC_PKCS12 structure pointer for new PKCS12 structure created. Can be NULL

Example:

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;

//bio loads in PKCS12 file
```

```
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, "a password", &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)

//use cert, pkey, and optionally certs stack
```

See Also:

wolfSSL_PKCS12_parse, wc_PKCS12_free

wolfSSL_set_tlsext_status_type

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_set_tlsext_status_type(WOLFSSL *s int type);

Description:

This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.

Return Values:

Return **1** on success or **0** on error.

Parameters:

s - pointer to WolfSSL struct which is created by SSL_new() function
type - ssl extension type which TLSEXT_STATUSTYPE_ocsp is only supported .

Example:

```
WOLFSSL *ssl;
WOLFSSL_CTX *ctx;
int ret;
```

```
ctx = wolfSSL_CTX_new(wolfSSLv23_server_method());
ssl = wolfSSL_new(ctx);
ret = WolfSSL_set_tlsext_status_type(ssl,TLSEXT_STATUSTYPE_ocsp);

wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
```

See Also:

wolfSSL_new
wolfSSL_CTX_new
wolfSSL_free
wolfSSL_CTX_free

wolfSSL_ASN1_TIME_adj

Synopsis:

include <wolfssl/ssl.h>

WOLFSSL_ASN1_TIME* wolfSSL_ASN1_TIME_adj(WOLFSSL_ASN1_TIME *s, time_t t, int offset_day, long offset_sec)

Description:

This function sets the ASN1_TIME structure s to the time represented by the time offset_day and offset_sec after the time_t value t.
if s is NULL, a new ASN1_TIME structure s is allocated and returned.

Return Values:

Returns a pointer to WOLFSSL_ASN1_TIME structure.

Parameters:

s - pointer to WOLFSSL_ASN1_TIME structure.
t - time_t time information to adjust.
offset_day - a number of days to adjust time_t t.
offset_sec - a number of secs to dajust timet_t t.

Example:

```
#include <wolfssl/ssl.h>

WOLFSSL_ASN_TIME *s,*adj_ret;
time_t t = 30 * years + 45 * days;
int offset_day = 10;
long offset_sec = 1200;

s = (WOLFSSL_ASN1_TIME *)malloc(sizeof(WOLFSSL_ASN1_TIME));
adj_ret = wolfSSL_ASN1_TIME_adj(s, t, offset_day, offset_sec);
```

See Also:

wolfSSL_X509_STORE_CTX_set_time

Synopsis:

#include <wolfssl/ssl.h>
void wolfSSL_X509_STORE_CTX_set_time(WOLFSSL_X509_STORE_CTX *ctx,unsigned long flags, time_t t)

Description:

This function sets certificate validation date.

Return Values:

No value returned.

Parameters:

ctx - pointer to WOLFSSL_X509_STORE_CTX.if NULL is passed, function allocate memory and return it.
flags - not used
time_t - time to validate certificate.

Example:

```
WOLFSSL_X509_STORE_CTX*  ctx;
time_t ctime;

ctx = XMALLOC(sizeof(WOLFSSL_X509_STORE_CTX),
NULL,DYNAMIC_TYPE_TMP_BUFFER);
ctx->param = XMALLOC(sizeof(WOLFSSL_X509_VERIFY_PARAM), NULL,
DYNAMIC_TYPE_TMP_BUFFER);
ctime = time_to validate;
wolfSSL_X509_STORE_CTX_set_time(ctx, 0, ctime);
```

See Also:

wolfSSL_X509_STORE_CTX_set_verify_cb

Synopsis:

```
#include <wolfssl/ssl.h>
void wolfSSL_X509_STORE_CTX_set_verify_cb(WOLFSSL_X509_STORE_CTX *ctx,
WOLFSSL_X509_STORE_CTX_verify_cb verify_cb)
```

Description:

This function sets the verification callback of ctx.

Callback prototype:
typedef void *WOLFSSL_X509_STORE_CTX_verify_cb;

Return Values:

No value returned.

Parameters:

ctx - pointer to WOLFSSL_X509_STORE_CTX
cb - verification callback function.

Example:

```
static int cb(int v, WOLFSSL_X509_STORE_CTX*ctx)
{ …
   return 1;
}


WOLFSSL_X509_STORE_CTX *ctx;

wolfSSL_X509_STORE_CTX_set_verify_cb(ctx, cb) ;
```

See Also:

wolfSSL_CTX_add_client_CA

Synopsis:

```
#include <wolfssl/ssh.h>
int wolfSSL_CTX_add_client_CA(WOLFSSL_CTX *ctx, WOLFSSL_X509 *x509)
```

Description:

This function adds client certificates to WOLFSSL_CTX context structure.

Return Values:

On success a SSL_SUCCESS is returned, on failure SSL_FAILURE is returned.

Parameters:

ctx - pointer to WOLFSSL_CTX structure to set client certificate in.
x509 - pointer to WOLFSSL_X509 structure which is client certificate information.

Example:

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
```

int ret;

ctx = wolfSSL_CTX_new(wolfSSLv23_client_method());
x509 = wolfSSL_X509_load_certificate_file(certfile, SSL_FILETYPE_PEM);
ret = wolfSSL_CTX_add_client_CA(ctx, x509);

See Also:

wolfSSL_X509_load_certificate_file
wolfSSL_SSL_CTX_get_cliet_CA_list

wolfSSL_CTX_set_srp_username

Synopsys:

#include <wolfssl/ssl.h>
int wolfSSL_CTX_set_srp_username(WOLFSSL_CTX* ctx, char* password)

Description:

This function set user name for SRP in WOLFSSL_CTX structure.

Return Values:

On success a SSL_SUCCESS is returned, on failure SSL_FAILURE is returned.

Parameters:

ctx - pointer to WOLFSSL_CTX_structure.
username - user name for SRP.

Examples:

WOLFSSL_CTX *ctx;
const char *username = "TESTUSER";
int r;

ctx = wolfSSL_CTX_new(wolfSSLv23_client_method());
r = wolfSSL_CTX_set_srp_username(ctx, (char *)username);

See Also:

wolfSSL_CTX_new
wolfSSL_CTX_set_srp_password

wolfSSL_CTX_set_srp_password

Synopsis:

```
#include <wolfssl/ssl.h>
int wolfSSL_CTX_set_srp_password(WOLFSSL_CTX* ctx, char* password)
```

Description:

This function sets password for SRP in WOLFSSL_CTX structure.

Return Values:

On success a SSL_SUCCESS is returned, on failure SSL_FAILURE is returned.

Parameters:

ctx - pointer to WOLFSSL_CTX structure.
password - password for SRP.

Example:

```
WOLFSSL_CTX *ctx;
const char *password = "TESTPASS";
int r;

ctx = wolfSSL_CTX_new(wolfSSLv23_client_method());
r = wolfSSL_CTX_set_srp_password(ctx, (char *)password);
```

See Also:

wolfSSL_CTX_new
wolfSSL_CTX_set_srp_username

wolfSSL_SSL_CTX_set_alpn_protos

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_set_alpn_protos(WOLFSSL_CTX *ctx, const unsigned char *p,
unsigned int p_len)

Description:

This function is used by the client to set the list of protocols available to be negotiated.
.

Parameters:

ctx - pointer to WOLFSSL_CTX structure.
P - protocol list in protocol-list format
P_len - list length

Example:

```
WOLFSSL_CTX *ctx;
unsigned char protos[] = {
     7, 't', 'l', 's', '/', '1', '.', '3',
     8, 'h', 't', 't', 'p', '/', '2', '.', '0'
};
unsigned int len = sizeof(protos);

SSL_CTX_set_alpn_protos(ctx, protos, len);
```

See Also:

wolfSSL_CTX_new

wolfSSL_d2i_X509_fp

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_X509* wolfSSL_d2i_X509_fp(XFILE fp, WOLFSSL_X509 **x509)

Description:

This function read DAR format certificate file and decode it to x509 parameter.
It returns pointer of WOLFSSL_X509 structure.
If error is occurred, it returns NULL,
.

Return Values:

NULL: Error
Not Null: Valid WOLFSSL_X509 pointer

Parameters:

fp - pointer of file structure
X509 - x509 certificate data

Example:

```
XFILE fp;
WOLFSSL_X509 *x509;

fp = XFOPEN("ca-cert.dar", "rb");
x509 = wolfSSL_d2i_X509_fp(fp, NULL);
…
X509_free(x509);
XCLOSE(fp);
```

See Also:

wolfSSL_d2i_X509

wolfSSL_X509_print

Synopsis:

#include <wolfssl/ssl.h>

Int wolfSSL_X509_print(WOLFSSL_BIO* bio, WOLFSSL_X509 **x509)

Description:

This function outputs X509 structure data to output stream which is specified in bio. If error occurs, function returns 0, otherwise 1.

.

Return Values:

WOLFSSL_SUCCESS: success
WOLFSSL_FAILUER: failure

Parameters:

bio - WOLFSSL_BIO to write to
x509 - x509 certificate data

Example:

```
WOLFSSL_BIO *bio;
WOLFSSL_X509 *x509;

X509 = wolfSSL_X509_load_certificate_file(“svr_cert.crt”,SSL_FILETYPE_PEM);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
x509 = wolfSSL_x509_print(bio, x509);
…
wolfSSL_BIO_free(bio);
```

See Also:

wolfSSL_EVP_PKEY_size

Synopsis:

#include <wolfssl/openssl/evp.h>

int wolfSSL_EVP_PKEY_size(WOLFSSL_EVP_PKEY *pkey)

Description:

This function returns the maximum size of a signature in bytes
.
Return Values:

Maximum size of a signature in bytes.

Parameters:

pkey - Pointer of WOLFSSL_EVP_PKEY

Example:

See Also:

wolfSSL_i2d_X509_bio

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_i2d_X509_bio(WOLFSSL_BIO *bio, WOLFSSL_X509 *x509)

Description:

This function encodes the structure pointed to by x509 into DAR format.
.
Return Values:

Positive Value: Number of bytes successfully encoded
Negative Value: Error code

Parameters:

bio - WOLFSSL_BIO to write to

x509 - certificate data

Example:

See Also:
wolfSSL_i2d_X509

wolfSSL_RSA_verify

Synopsis:

#include <wolfssl/openssl/rsa.h>

int wolfSSL_RSA_verify(int type, const unsigned char *m, unsigned int m_len, unsigned char *sigbuf, unsigned int *siglen, RSA *rsa)

Description:

This function verifies that the signature sigbuf of size siglen matches a given message digests m of size m_len.
.

Return Values:

1: Success
0: Fail

Parameters:

type - Message digest algorithm
m - Message digest
m_len - Message digest length
sigbuf - Signature data
siglen - Signature length
rsa - Signer’s public key

Example:

See Also:

wolfSSL_X509_check_ca

Synopsis:

#include <wolfssl/openssl/ssl.h>

```
int wolfSSL_X509_check_ca(WOLFSSL_X509 *x509);
```

Description:

This function checks type of CA certificate.

Return Values:

0: Not CA certificate.

1: CA certificate.

4: Certificate with keyUsage extension.

WOLFSSL_FAILURE: Invalid argument

Parameters:

x509 - Certificate to be checked

Example:

See Also:

X509_verify_cert

wolfSSL_X509_CRL_free

Synopsis:

#include <wolfssl/openssl/ssl.h>

```
void wolfSSL_X509_CRL_free(WOLFSSL_X509_CRL *crl);
```

Description:

This function releases CRL pointer and its attached resources.

Return Values:

None

Parameters:

crl - CRL pointer

Example:

See Also:

wolfSSL_X509_CRL_new

wolfSSL_d2i_PKCS12_fp

Synopsis:

#include <wolfssl/openssl/ssl.h>

WOLFSSL_X509_PKCS12* wolfSSL_d2i_PKCS12_fp(XFILE fp, WOLFSSL_X509_PKCS12** pkcs12);

Description:

This function reads X509 PKCS12 from given file and converts to internal format. New pointer is created for the converted PKCS12 internal format. If pkcs12 pointer, the pointer is also returned as a return value.

Return Values:

Not NULL: PKCS12 pointer
NULL: Invalid argument, process error

Parameters:

fp - Input file pointer

pkcs12 - Pointer to PKCS12 pointer

Example:

See Also:

wolfSSL_d2i_PKCS12_bio

wolfSSL_d2i_X509_CRL_fp

Synopsis:

#include <wolfssl/openssl/ssl.h>

WOLFSSL_X509_CRL *wolfSSL_d2i_X509_CRL_fp(XFILE fp, WOLFSSL_X509_CRL **crl)

Description:

This function reads X509 CRL from given file and converts to internal format. New pointer is created for the converted CRL internal format. If CRL pointer, the pointer is also returned as a return value.

Return Values:

Not NULL: CRL pointer
NULL: Invalid argument, process error

Parameters:

fp - Input file pointer

crl - Pointer to CRL pointer

Example:

See Also:

wolfSSL_d2i_X509_CRL, wolfSSL_d2i_PKCS12_fp,

wolfSSL_i2d_RSAPublicKey

Synopsis:

#include <wolfssl/openssl/ssl.h>

int wolfSSL_i2d_RSAPublicKey(WOLFSSL_RSA *rsa, const unsigned char **pp);

Description:

This function converts internal format RSA public key to DER format. New pointer is created for DER format. The pointer is also returned as a return value.

Return Values:

Not NULL: DER pointer
NULL: Invalid argument, process error

Parameters:

rsa - RSA public key pointer

pp - Pointer of pointer to DER format

Example:

See Also:

wolfSSL_d2i_RSAPublicKey

wolfSSL_d2i_RSAPublicKey

Synopsis:

#include <wolfssl/openssl/ssl.h>

WOLFSSL_RSA *wolfSSL_d2i_RSAPublicKey(WOLFSSL_RSA **rsa, const unsigned char **der, long len);

Description:

This function converts DER format public key to internal format RSA key. New pointer is created for RSA. The pointer is also returned as a return value.

Return Values:

Not NULL: RSA pointer
NULL: Invalid argument, process error

Parameters:

rsa - Pointer to RSA public key pointer

der - Pointer to DER pointer

len - DER length

Example:

See Also:

wolfSSL_i2d_RSAPublicKey

wolfSSL_PEM_read_X509_CRL

Synopsis:

#include <wolfssl/openssl/ssl.h>

```
WOLFSSL_X509_CRL* wolfSSL_PEM_read_X509_CRL(XFILE fp, WOLFSSL_X509_CRL
**crl, pem_password_cb *cb, void *u);
```

Description:

This function reads CRL file into internal CRL format. A CRL pointer is created.

Return Values:

Non NULL: Created CRL poiter
NULL: Process Error

Parameters:

fp - File pointer

crl - Pointer to created CRL pointer

cb - void

u - void

Example:

See Also:

wolfSSL_PEM_read_X509

wolfSSL_X509_STORE_add_crl

Synopsis:

#include <wolfssl/openssl/ssl.h>

int wolfSSL_X509_STORE_add_crl(WOLFSSL_X509_STORE *store, WOLFSSL_X509_CRL *newcrl);

Description:

This function adds CRL to X509 Store.

Return Values:

Negative Value: Error
Non-negative Value: Success

Parameters:

crl - CRL pointer

Example:

See Also:

wolfSSL_X509_CRL_new, wolfSSL_X509_new

wolfSSL_OPENSSL_add_all_algorithms_noconf

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_OPENSSL_add_all_algorithms_noconf(void);

Description:

This function adds all algorithms to the crypto core, but don't use the configuration file.

Return Values:

WOLFSSL_SUCCESS: Success
WOLFSSL_FATAL_ERROR: Error

Parameters:

Example:

See Also:

wolfSSL_add_all_algorithms

wolfSSL_ASN1_GENERALIZEDTIME_free

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_ASN1_GENERALIZEDTIME_free(WOLFSSL_ASN1_GENERALIZEDTIME* gtime);

Description:

This function frees and clears WOLFSSL_ASN1_TIME data.

Return Values:

Parameters:

gtime: Pointer of WOLFSSL_ASN1_TIME

Example:

See Also:

wolfSSL_ASN1_GENERALIZEDTIME_print

wolfSSL_RAND_poll

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_RAND_poll(void);

Description:

This function generates seed using random generator.

Return Values:

WOLFSSL_SUCCESS: Success
WOLFSSL_FAILURE: Failure

Parameters:

Example:

See Also:

wolfSSL_RAND_seed

wolfSSL_ASN1_STRING_print_ex

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_ASN1_STRING_print_exl(WOLFSSL_BIO *bio, WOLFSSL_ASN1_STRING *str, unsigned long flags);

Description:

This function outputs ASN1_STRING structure.

Return Values:

Positive Value: Length of output string.

WOLFSSL_FAILURE: Failure

Parameters:

bio - WOLFSSL_BIO to write to

str - WOLFSSL_ASN!_STRING data

flags - Flags which indcates Output format flags

ASN1_STRFLGS_SHOW_TYPE: Output type information

ASN1_STRFLGS_DUMP_ALL: Output hex dump

ASN1_STRFLGS_DUMP_DER: Output hex dump for DER format

ASN1_STRFLGS_ESC_2253: Escape the character which is specified by RFC2253

Example:

See Also:

wolfSSL_X509_NAME_print_ex, wolfSSL_ASN1_tag2str

wolfSSL_ASN1_tag2str

Synopsis:

```
#include <wolfssl/ssl.h>

const char* wolfSSL_ASN1_tag2str(int tag);
```

Description:

This function returns a human-readable name of the specified ASN.1 tag.

Return Values:

ASN1 tag string

Parameters:

tag - ASN1 tag type

Example:

See Also:

wolfSSL_ASN1_STRING_print_ex

wolfSSL_ASN1_TIME_to_generalizedtime

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_ASN1_TIME* wolfSSL_ASN1_TIME_to_generalizedtime(WOLFSSL_ASN1_TIME *t, WOLFSSL_ASN1_TIME **out);

Description:

This function converts WOLFSSL_ASN1_TIME to ASN1 GENERALIZEDTIME.

Return Values:

Not NULL: WOLFSSL_ASN1TIME data which is converted generalized time format.
NULL: Error

Parameters:

t - ASN1_TIME data

out - converted ASN1_TIME output data

Example:

See Also:

wolfSSL_ASN1_TIME_to_string, wolfSSL_ASN1_TIME_print

wolfSSL_X509_verify_cert

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_X509_verify_cert(WOLFSSL_X509_STORE_CTX *ctx);

Description:

This function verifies certificate chain in ctx.

Return Values:

WOLFSSL_SUCCESS: Success
WOLFSSL_FATAL_ERROR: Error

Parameters:

ctx - WOLFSSL_X509_STORE_CTX

Example:

See Also:

wolfSSL_i2c_ASN1_INTEGER

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_i2c_ASN1_INTEGER(WOLFSSL_ASN1_INTEGER *a, unsigned char **pp);

Description:

This function converts WOLFSSL_ASN1_INTEGER to inner representation.

Return Values:

WOLFSSL_SUCCESS: Success
WOLFSSL_FAILURE: Error

Parameters:

a - WOLFSSL_ASN1_INTEGER

pp - Converted ASN1 INTEGER data

Example:

See Also:

wolfSSL_X509_CA_num

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_X509_CA_num(WOLFSSL_X509_STORE *store);

Description:

This function counts certificates in WOLFSSL_X509_STORE.

Return Values:

Number of certificate in WOLFSSL-X509_STORE.

Parameters:

store - WOLFSSL_X509_STORE data

Example:

See Also:

wolfSSL_X509_get_version

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_X509_get_version(WOLFSSL_X509 *x509);

Description:

This function gets certificate’s X509 version.

Return Values:

Number of version - 1.

Parameters:

x509 - WOLFSSL_X509 data

Example:

See Also:

wolfSSL_ASN1_TIME_get_length

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_ASN1_TIME_get_length(WOLFSSL_ASN1_TIME *t);

Description:

This function return ASN1_TIME length.

Return Values:

Length of ASN1_TIME

Parameters:

t - WOLFSSL_ASN1_TIME data

Example:

See Also:

wolfSSL_ASN1_TIME_get_data

wolfSSL_ASN1_TIME_get_data

Synopsis:

#include <wolfssl/ssl.h>

```
int wolfSSL_ASN1_TIME_get_data(WOLFSSL_ASN1_TIME *t, unsigned char *data);
```

Description:

This function return ASN1_TIME data.

Return Values:

WOLFSSL_SUCCESS: Success

WOLFSSL_FAILURE: Error

Parameters:

t - WOLFSSL_ASN1_TIME

data - WOLFSSL_ASN1_TIME data

Example:

See Also:

wolfSSL_ASN1_TIME_get_length

wolfSSL_BIO_new_bio_pair

Synopsis:

#include <wolfssl/ssl.h>

```
int wolfSSL_BIO_new_bio_pair(WOLFSSL_BIO **bio1_p, size_t writebuf1,
                             WOLFSSL_BIO **bio2_p, size_t writebuf2)
```

Description:

This function creates a new bio pair.

This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

WOLFSSL_FAILURE: If the function fails at any point.

BAD_FUNC_ARG: If bio1_p or bio2_p is NULL.

WOLFSSL_SUCCESS: If the new bio pair is successfully created.

Parameters:

bio1_p: The passed in bio object to pair up with bio2_p.

bio2_p: The passed in bio object to pair up with bio1_p.

writebuf1: The size of the write buffer for bio1_p.

writebuf2: The size of the write buffer for bio2_p.

Example:

```
BIO* bio1 = NULL;
BIO* bio2 = NULL;

if(BIO_new_bio_pair(&bio1, 256, &bio2, 256) == WOLFSSL_SUCCESS) {
        ...
}
```

See Also:

wolfSSL_BIO_make_bio_pair

wolfSSL_d2i_RSAPrivateKey_bio

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_RSA* wolfSSL_d2i_RSAPrivateKey_bio(WOLFSSL_BIO *bio,
WOLFSSL_RSA **out)

Description:

This function converts an rsa key from a bio buffer into an internal rsa structure. This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

NULL: If anything in the function fails.

WOLFSSL_RSA* key: If the function executes correctly.

Parameters:

bio - The bio structure that should contain the rsa key buffer.

out - An optional parameter that can be set to NULL or can be used to store the resulting WOLFSSL_RSA key.

Example:

```
BIO* bio = NULL;
bio = BIO_new(BIO_s_mem());
BIO_write(bio, client_key_der_2048, sizeof_client_key_der_2048)

if(wolfSSL_d2i_RSAPrivateKey_bio(bio, NULL) != NULL) {
      ...
}
```

See Also:

wolfSSL_d2i_RSAPrivateKey, wolfSSL_d2i_PrivateKey_bio

wolfSSL_d2i_PrivateKey_bio

Synopsis:

#include <wolfssl/ssl.h>
WOLFSSL_EVP_PKEY* wolfSSL_d2i_PrivateKey_bio(WOLFSSL_BIO* bio,
WOLFSSL_EVP_PKEY** out)

Description:

This function converts EVP_PKEY data from a bio buffer to a WOLFSSL_EVP_PKEY structure.
This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

NULL: If anything in the function fails.

WOLFSSL_EVP_PKEY* key: If the function executes correctly.

Parameters:

bio - The bio structure that should contain an rsa key buffer or ecc key buffer.

out - An optional parameter that can be set to NULL or can be used to store the resulting WOLFSSL_EVP_PKEY key.

Example:

```
BIO* bio = NULL;
bio = BIO_new(BIO_s_mem());
BIO_write(bio, client_key_der_2048, sizeof_client_key_der_2048)

if(wolfSSL_d2i_PrivateKey_bio(bio, NULL) != NULL) {
       ...
}
```

See Also:

wolfSSL_d2i_PrivateKey_EVP, wolfSSL_d2i_PUBKEY_bio, wolfSSL_d2i_RSAPrivateKey_bio

wolfSSL_CTX_use_certificate_ASN1

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_CTX_use_certificate_ASN1(WOLFSSL_CTX *ctx, int derSz,
const unsigned char *der)

Description:

This function loads the ASN1 type certificate used in the SSL connection into the SSL context (WOLFSSL_CTX).
This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

WOLFSSL_FAILURE: If the ctx or der parameters are NULL or loading the certificate fails.

WOLFSSL_SUCCESS: If the ASN1 certificate is successfully loaded into the the SSL context.

Parameters:

ctx: Pointer to the ssl context.

der: The der encoded ASN1 certificate.

derSz: The size of the der encoded ASN1 certificate.

Example:
```
WOLFSSL_CTX* ctx;
int ret;

ret = SSL_CTX_use_certificate_ASN1(ctx, sizeof_server_cert_der_2048,
                server_cert_der_2048);
if(ret == WOLFSSL_SUCCESS) {
      ...
}
```

See Also:
wolfSSL_CTX_use_certificate_buffer

wolfSSL_CTX_use_RSAPrivateKey

Synopsis:
#include <wolfssl/ssl.h>
int wolfSSL_CTX_use_RSAPrivateKey(WOLFSSL_CTX* ctx, WOLFSSL_RSA* rsa)

Description:
This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX).
This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:
BAD_FUNC_ARG: If the CTX or RSA parameter is passed in as NULL.

WOLFSSL_FAILURE: If the functions fails at any point during execution.

WOLFSS_SUCCESS: If the the function executes successfully.

Parameters:
ctx: Pointer to an CTX structure.

rsa: Pointer to the RSA structure.

Example:
```
int ret;
WOLFSSL_CTX* ctx;
WOLFSSL_RSA* rsa;

...

ret = wolfSSL_CTX_use_RSAPrivateKey(ctx, rsa);
if (ret != SSL_SUCCESS) {
      // error loading private key
}

...
```

See Also:
wolfSSL_CTX_use_RSAPrivateKey_file

wolfSSL_d2i_PrivateKey_bio

Synopsis:
#include <wolfssl/ssl.h>
WOLFSSL_EVP_PKEY* wolfSSL_d2i_PrivateKey_bio(WOLFSSL_BIO* bio,
WOLFSSL_EVP_PKEY** out)

Description:
This function extracts EVP_PKEY data such as an RSA or ECC private key from a BIO object and creates a WOLFSSL_EVP_PKEY structure containing the given key data. Extra Information in the BIO structure that is not the key data is kept in the BIO structure while the key is removed.
This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:
NULL: If the function fails at any point while executing.

WOLFSSL_EVP_PKEY: If the function executes successfully then it will return the PKEY structure containing the private key.

Parameters:
ctx - WOLFSSL_CTX structure to set user argument.

arg - user argument.

Example:

```
BIO* bio = NULL;
WOLFSSL_EVP_PKEY pkey = NULL;

//write private key to bio

pkey = wolfSSL_d2i_PrivateKey_bio(bio, NULL);
if (pkey != NULL) {
        ...
}
```

See Also:

wolfSSL_d2i_PrivateKey, wolfSSL_d2i_PrivateKey_EVP, wolfSSL_d2i_PUBKEY

wolfSSL_d2i_PrivateKey_EVP

Synopsis:

#include <wolfssl/ssl.h>

WOLFSSL_EVP_PKEY* wolfSSL_d2i_PrivateKey_EVP(WOLFSSL_EVP_PKEY** out,
unsigned char** in, long inSz)

Description:

This functions converts a DER encoded private key into a WOLFSSL_EVP_PKEY structure. It can be passed in an RSA private key or an ECC private key and is capable of determining which type it is.

This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

NULL: If the function fails at any point.

WOLFSSL_EVP_PKEY: If the function executes successfully then it will return the PKEY structure containing the private key.

Parameters:

out: An optional paramter that can be used to store the resulting WOLFSSL_EVP_PKEY structure containing the private key. Can be NULL if desired.

In: The passed in DER encoded private key buffer to be placed in the PKEY structure.

inSz: The size of the passed in private key buffer.

Example:

```
unsigned char* mem = NULL;
int memSz;
WOLFSSL_EVP_PKEY* key = NULL;
BIO* bio = NULL;
...
memSz = wolfSSL_BIO_pending(bio);
...
mem = (unsigned char*)XMALLOC(memSz, bio->heap, DYNAMIC_TYPE_TMP_BUFFER);
...

if (wolfSSL_BIO_read(bio, (unsigned char*)mem, memSz) == memSz) {
    if ((key = wolfSSL_d2i_PrivateKey_EVP(NULL, &mem, (long)memSz)) == NULL)
    {
            //error, free memory and return NULL
    }
}
```

See Also:

wolfSSL_d2i_PrivateKey, wolfSSL_d2i_PrivateKey_bio, wolfSSL_d2i_PUBKEY

wolfSSL_d2i_RSAPrivateKey

Synopsis:

#include <wolfssl/ssl.h>
WOLFSSL_RSA *wolfSSL_d2i_RSAPrivateKey(WOLFSSL_RSA **r,
const unsigned char **derBuf, long derSz)

Description:

This function converts a der encoded buffer into a WOLFSSL_RSA structure.
This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

NULL: If a step in the function fails.

WOLFSSL_RSA: If the function executes successfully then it will return the new WOLFSSL_RSA structure that contains the RSA private key.

Parameters:

r: An optional parameter that can be used to store the resulting WOLFSSL_RSA structure containing the RSA private key. Can be set to null if the return value is being used instead.

derBuf: The der encoded buffer containing the rsa private key.

derSz: The size of the buffer containing the rsa private key.

Example:

```
unsigned char* mem = NULL;
int memSz;
WOLFSSL_RSA* key = NULL;
BIO* bio = NULL;
...
memSz = wolfSSL_BIO_pending(bio);
...
mem = (unsigned char*)XMALLOC(memSz, bio->heap, DYNAMIC_TYPE_TMP_BUFFER);
...

if (wolfSSL_BIO_read(bio, (unsigned char*)mem, memSz) == memSz) {
    if ((key = wolfSSL_d2i_RSAPrivateKey(NULL, &mem, (long)memSz)) == NULL)
    {
            //error, free memory and return NULL
    }
}
```

See Also:

wolfSSL_d2i_RSAPrivateKey, wolfSSL_d2i_RSAPrivateKey_bio

wolfSSL_i2d_RSAPrivateKey

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_i2d_RSAPrivateKey(WOLFSSL_RSA *rsa, unsigned char **pp)

Description:

This function extracts an RSA private key from a WOLFSSL_RSA structure and stores it in a DER type buffer.

This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

On success, this function will return the size of the der buffer containing the RSA key.

WOLFSSL_FAILURE: If any of the internal function calls fail.

BAD_FUNC_ARG: If rsa or pp are equal to NULL.

Parameters:

rsa - The rsa structure to pass in.

pp - The buffer that will contain the rsa key if the function successfully executes.

Example:

```
unsigned char maxDerBuf[4096];
unsigned char* key = NULL;
WOLFSSL_RSA* rsa = NULL;
…

key = maxDerBuf;
if((derSize = wolfSSL_i2d_RSAPrivateKey(rsa, &key)) <= 0) {
      //error, return failure.
}
```

See Also:

wolfSSL_d2i_RSAPrivateKey

wolfSSL_get_verify_callback

Synopsis:

#include <wolfssl/ssl.h>

VerifyCallback wolfSSL_get_verify_callback(WOLFSSL* ssl)

Description:

This function returns the verifyCallback from the ssl structure if successful.
This function is only available when wolfSSL has been configured with the OpenSSL compatibility layer enabled (--enable-opensslall) or when the asio compatibility is enabled (--enable-asio).

Return Values:

NULL: If the passed in ssl structure is NULL.

ssl->verifyCallback: If the method successfully executes, the verifyCallback object is returned.

Parameters:

ssl - Pointer to the SSL structure.

Example:
```
WOLFSSL_SSL* ssl;

// setup ssl

if(wolfSSL_get_verify_callback(ssl) != NULL) {
      ...
}
```

See Also:
wolfSSL_CTX_get_verify_callback

17.13 TLS Extensions

The functions in this section are specific to supported TLS extensions.

wolfSSL_CTX_UseSNI

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseSNI(WOLFSSL_CTX* ctx, byte type,
                                        const void* data, word16 size);

Description:
This function enables the use of Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL clients and wolfSSL servers will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
* ctx is NULL
* data is NULL
* type is a unknown value. (see below)

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ctx - pointer to a SSL context, created with wolfSSL_CTX_new().

type - indicates which type of server name is been passed in data. The known types are:
enum {
WOLFSSL_SNI_HOST_NAME = 0
};

data - pointer to the server name data.

size - size of the server name data.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ret = wolfSSL_CTX_UseSNI(ctx, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
strlen("www.yassl.com"));

if (ret != 0) {
    // sni usage failed
}
```

See Also:

wolfSSL_CTX_new
wolfSSL_UseSNI

wolfSSL_UseSNI

Synopsis:
#include <wolfssl/ssl.h>

```
int wolfSSL_UseSNI(WOLFSSL* ssl, unsigned char type,
                   const void* data, word16 size);
```

Description:
This function enables the use of Server Name Indication in the SSL object passed in the 'ssl' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL client and wolfSSL server will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
* ssl is NULL
* data is NULL
* type is a unknown value. (see below)

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ssl - pointer to a SSL object, created with wolfSSL_new().

type - indicates which type of server name is been passed in data. The known types are:

```
enum {
    WOLFSSL_SNI_HOST_NAME = 0
};
```

data - pointer to the server name data.

size - size of the server name data.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);

if (ssl == NULL) {
    // ssl creation failed
}

ret = wolfSSL_UseSNI(ssl, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
strlen("www.yassl.com"));

if (ret != 0) {
    // sni usage failed
}
```

See Also:
wolfSSL_new
wolfSSL_CTX_UseSNI

wolfSSL_CTX_SNI_SetOptions

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_CTX_SNI_SetOptions(WOLFSSL_CTX* ctx, byte type,
unsigned char options);

Description:
This function is called on the server side to configure the behavior of the SSL sessions using Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. The options are explained below.

Return Values:

This function does not have a return value.

Parameters:

ctx - pointer to a SSL context, created with wolfSSL_CTX_new().

type - indicates which type of server name is been passed in data. The known types are:

```
enum {
    WOLFSSL_SNI_HOST_NAME = 0
};
```

options - a bitwise semaphore with the chosen options. The available options are:

```
enum {
    WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01,
    WOLFSSL_SNI_ANSWER_ON_MISMATCH   = 0x02
};
```

Normally the server will abort the handshake by sending a fatal-level unrecognized_name(112) alert if the hostname provided by the client mismatch with the servers.

WOLFSSL_SNI_CONTINUE_ON_MISMATCH - With this option set, the server will not send a SNI response instead of aborting the session.

WOLFSSL_SNI_ANSWER_ON_MISMATCH - With this option set, the server will send a SNI response as if the host names match instead of aborting the session.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ret = wolfSSL_CTX_UseSNI(ctx, 0, "www.yassl.com", strlen("www.yassl.com"));

if (ret != 0) {
```

```
    // sni usage failed
}

wolfSSL_CTX_SNI_SetOptions(ctx, WOLFSSL_SNI_HOST_NAME,
WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

See Also:
wolfSSL_CTX_new
wolfSSL_CTX_UseSNI
wolfSSL_SNI_SetOptions

wolfSSL_SNI_SetOptions

Synopsis:
#include <wolfssl/ssl.h>

void wolfSSL_SNI_SetOptions(WOLFSSL* ssl, unsigned char type, unsigned char options);

Description:
This function is called on the server side to configure the behavior of the SSL session using Server Name Indication in the SSL object passed in the 'ssl' parameter. The options are explained below.

Return Values:
This function does not have a return value.

Parameters:

ssl - pointer to a SSL object, created with wolfSSL_new().

type - indicates which type of server name is been passed in data. The known types are:

```
enum {
    WOLFSSL_SNI_HOST_NAME = 0
};
```

options - a bitwise semaphore with the chosen options. The available options are:

```
enum {
    WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01,
```

```
    WOLFSSL_SNI_ANSWER_ON_MISMATCH   = 0x02
};
```

Normally the server will abort the handshake by sending a fatal-level unrecognized_name(112) alert if the hostname provided by the client mismatch with the servers.

WOLFSSL_SNI_CONTINUE_ON_MISMATCH - With this option set, the server will not send a SNI response instead of aborting the session.

WOLFSSL_SNI_ANSWER_ON_MISMATCH - With this option set, the server will send a SNI response as if the host names match instead of aborting the session.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);

if (ssl == NULL) {
    // ssl creation failed
}

ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));

if (ret != 0) {
    // sni usage failed
}

wolfSSL_SNI_SetOptions(ssl, WOLFSSL_SNI_HOST_NAME,
WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

See Also:
wolfSSL_new
wolfSSL_UseSNI
wolfSSL_CTX_SNI_SetOptions

wolfSSL_SNI_GetRequest

Synopsis:
#include <wolfssl/ssl.h>

word16 wolfSSL_SNI_GetRequest(WOLFSSL *ssl, byte type, void** data);

Description:
This function is called on the server side to retrieve the Server Name Indication provided by the client in a SSL session.

Return Values:
The size of the provided SNI data.

Parameters:

ssl - pointer to a SSL object, created with wolfSSL_new().

type - indicates which type of server name is been retrieved in data. The known types are:

```
enum {
    WOLFSSL_SNI_HOST_NAME = 0
};
```

data - pointer to the data provided by the client.

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);

if (ssl == NULL) {
    // ssl creation failed
```

```
}

ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));

if (ret != 0) {
    // sni usage failed
}

if (wolfSSL_accept(ssl) == SSL_SUCCESS) {
    void *data = NULL;
    unsigned short size = wolfSSL_SNI_GetRequest(ssl, 0, &data);
}
```

See Also:
wolfSSL_UseSNI
wolfSSL_CTX_UseSNI

wolfSSL_SNI_GetFromBuffer

Synopsis:
#include <wolfssl/ssl.h>

WOLFSSL_API int wolfSSL_SNI_GetFromBuffer(const byte* clientHello, word32 helloSz, byte type, byte* sni, word32* inOutSz);

Description:
This function is called on the server side to retrieve the Server Name Indication provided by the client from the Client Hello message sent by the client to start a session. It does not requires context or session setup to retrieve the SNI.

Return Values:
If successful the call will return **SSL_SUCCESS**;
If there is no SNI extension in the client hello, the call will return **0**.

BAD_FUNC_ARG is the error that will be returned in one of this cases:
* buffer is NULL
* bufferSz <= 0
* sni is NULL
* inOutSz is NULL or <= 0

BUFFER_ERROR is the error returned when there is a malformed Client Hello message.

INCOMPLETE_DATA is the error returned when there is not enough data to complete the extraction.

Parameters:

buffer - pointer to the data provided by the client (Client Hello).

bufferSz - size of the Client Hello message.

type - indicates which type of server name is been retrieved from the buffer. The known types are:
  enum {
    WOLFSSL_SNI_HOST_NAME = 0
  };

sni - pointer to where the output is going to be stored.

inOutSz - pointer to the output size, this value will be updated to MIN("SNI's length", inOutSz).

Example:

```
unsigned char buffer[1024] = {0};
unsigned char result[32]   = {0};
int           length       = 32;

// read Client Hello to buffer...

ret = wolfSSL_SNI_GetFromBuffer(buffer, sizeof(buffer), 0, result, &length));
if (ret != SSL_SUCCESS) {
    // sni retrieve failed
}
```

See Also:
wolfSSL_UseSNI
wolfSSL_CTX_UseSNI
wolfSSL_SNI_GetRequest

wolfSSL_CTX_UseMaxFragment

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseMaxFragment(WOLFSSL_CTX* ctx, byte mfl);

Description:
This function is called on the client side to enable the use of Maximum Fragment Length for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
* ctx is NULL
* mfl is out of range.

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ctx - pointer to a SSL context, created with wolfSSL_CTX_new().

mfl - indicates which is the Maximum Fragment Length requested for the session. The available options are:

```
enum {
    WOLFSSL_MFL_2_9  = 1, /*  512 bytes */
    WOLFSSL_MFL_2_10 = 2, /* 1024 bytes */
    WOLFSSL_MFL_2_11 = 3, /* 2048 bytes */
    WOLFSSL_MFL_2_12 = 4, /* 4096 bytes */
    WOLFSSL_MFL_2_13 = 5  /* 8192 bytes *//* wolfSSL ONLY!!! */
};
```

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);
```

```
if (ctx == NULL) {
    // context creation failed
}

ret = wolfSSL_CTX_UseMaxFragment(ctx, WOLFSSL_MFL_2_11);

if (ret != 0) {
    // max fragment usage failed
}
```

See Also:
wolfSSL_CTX_new
wolfSSL_UseMaxFragment

wolfSSL_UseMaxFragment

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_UseMaxFragment(WOLFSSL* ssl, byte mfl);

Description:
This function is called on the client side to enable the use of Maximum Fragment Length in the SSL object passed in the 'ssl' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
  * ssl is NULL
  * mfl is out of range.

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ssl - pointer to a SSL object, created with wolfSSL_new().

mfl - indicates witch is the Maximum Fragment Length requested for the session. The available options are:

```
enum {
    WOLFSSL_MFL_2_9  = 1, /*  512 bytes */
    WOLFSSL_MFL_2_10 = 2, /* 1024 bytes */
    WOLFSSL_MFL_2_11 = 3, /* 2048 bytes */
    WOLFSSL_MFL_2_12 = 4, /* 4096 bytes */
    WOLFSSL_MFL_2_13 = 5  /* 8192 bytes *//* wolfSSL ONLY!!! */
};
```

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);

if (ssl == NULL) {
    // ssl creation failed
}

ret = wolfSSL_UseMaxFragment(ssl, WOLFSSL_MFL_2_11);

if (ret != 0) {
    // max fragment usage failed
}
```

See Also:
wolfSSL_new
wolfSSL_CTX_UseMaxFragment

wolfSSL_CTX_UseTruncatedHMAC

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseTruncatedHMAC(WOLFSSL_CTX* ctx);

Description:

This function is called on the client side to enable the use of Truncated HMAC for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

Return Values:

If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
  * ctx is NULL

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ctx - pointer to a SSL context, created with wolfSSL_CTX_new().

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ret = wolfSSL_CTX_UseTruncatedHMAC(ctx);

if (ret != 0) {
    // truncated HMAC usage failed
}
```

See Also:

wolfSSL_CTX_new
wolfSSL_UseMaxFragment

wolfSSL_UseTruncatedHMAC

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_UseTruncatedHMAC(WOLFSSL* ssl);

Description:
This function is called on the client side to enable the use of Truncated HMAC in the SSL object passed in the 'ssl' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
 * ssl is NULL

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ssl - pointer to a SSL object, created with wolfSSL_new().

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);

if (ssl == NULL) {
    // ssl creation failed
}

ret = wolfSSL_UseTruncatedHMAC(ssl);
```

```
if (ret != 0) {
    // truncated HMAC usage failed
}
```

See Also:
wolfSSL_new
wolfSSL_CTX_UseMaxFragment

wolfSSL_CTX_UseSupportedCurve

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_CTX_UseSupportedCurve(WOLFSSL_CTX* ctx, word16 name);

Description:
This function is called on the client side to enable the use of Supported Elliptic Curves Extension for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.
Return Values:
If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
* ctx is NULL
* name is a unknown value. (see below)

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ctx - pointer to a SSL context, created with wolfSSL_CTX_new().

name - indicates which curve will be supported for the session. The available options are:
enum {
WOLFSSL_ECC_SECP160R1 = 0x10,
WOLFSSL_ECC_SECP192R1 = 0x13,

```
        WOLFSSL_ECC_SECP224R1 = 0x15,
        WOLFSSL_ECC_SECP256R1 = 0x17,
        WOLFSSL_ECC_SECP384R1 = 0x18,
        WOLFSSL_ECC_SECP521R1 = 0x19
    };
```

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ret = wolfSSL_CTX_UseSupportedCurve(ctx, WOLFSSL_ECC_SECP256R1);

if (ret != 0) {
    // Elliptic Curve Extension usage failed
}
```

See Also:

wolfSSL_CTX_new
wolfSSL_UseSupportedCurve

wolfSSL_UseSupportedCurve

Synopsis:

#include <wolfssl/ssl.h>

int wolfSSL_UseSupportedCurve(WOLFSSL* ssl, word16 name);

Description:

This function is called on the client side to enable the use of Supported Elliptic Curves Extension in the SSL object passed in the 'ssl' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.

Return Values:

If successful the call will return **SSL_SUCCESS**.

BAD_FUNC_ARG is the error that will be returned in one of these cases:
* ssl is NULL
* name is a unknown value. (see below)

MEMORY_E is the error returned when there is not enough memory.

Parameters:

ssl - pointer to a SSL object, created with wolfSSL_new().

name - indicates which curve will be supported for the session. The available options are:

```
enum {
    WOLFSSL_ECC_SECP160R1 = 0x10,
    WOLFSSL_ECC_SECP192R1 = 0x13,
    WOLFSSL_ECC_SECP224R1 = 0x15,
    WOLFSSL_ECC_SECP256R1 = 0x17,
    WOLFSSL_ECC_SECP384R1 = 0x18,
    WOLFSSL_ECC_SECP521R1 = 0x19
};
```

Example:

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;

ctx = wolfSSL_CTX_new(method);

if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);

if (ssl == NULL) {
    // ssl creation failed
}

ret = wolfSSL_UseSupportedCurve(ssl, WOLFSSL_ECC_SECP256R1);

if (ret != 0) {
```

```
    // Elliptic Curve Extension usage failed
}
```

See Also:
wolfSSL_CTX_new
wolfSSL_CTX_UseSupportedCurve

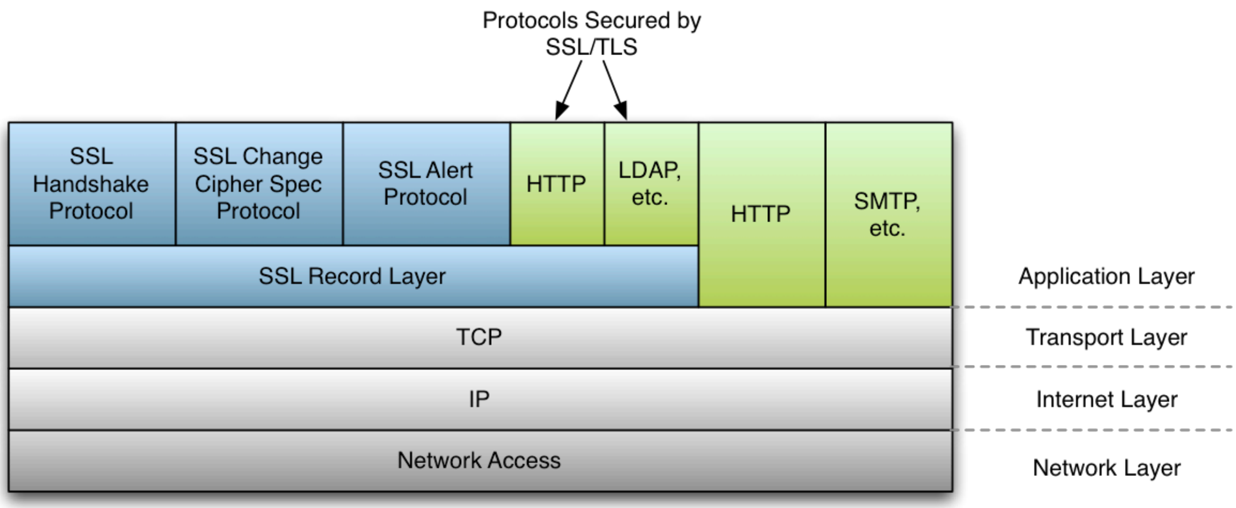
Appendix A: SSL/TLS Overview

A.1 General Architecture

The wolfSSL (formerly CyaSSL) embedded SSL library implements SSL 3.0, TLS 1.0, TLS 1.1 and TLS 1.2 protocols. TLS 1.2 is currently the most secure and up to date version of the standard. wolfSSL does not support SSL 2.0 due to the fact that it has been insecure for several years.

The TLS protocol in wolfSSL is implemented as defined in [RFC 5246 (http://tools.ietf.org/html/rfc5246).](http://tools.ietf.org/html/rfc5246) Two record layer protocols exist within SSL - the message layer and the handshake layer. Handshake messages are used to negotiate a common cipher suite, create secrets, and enable a secure connection. The message layer encapsulates the handshake layer while also supporting alert processing and application data transfer.

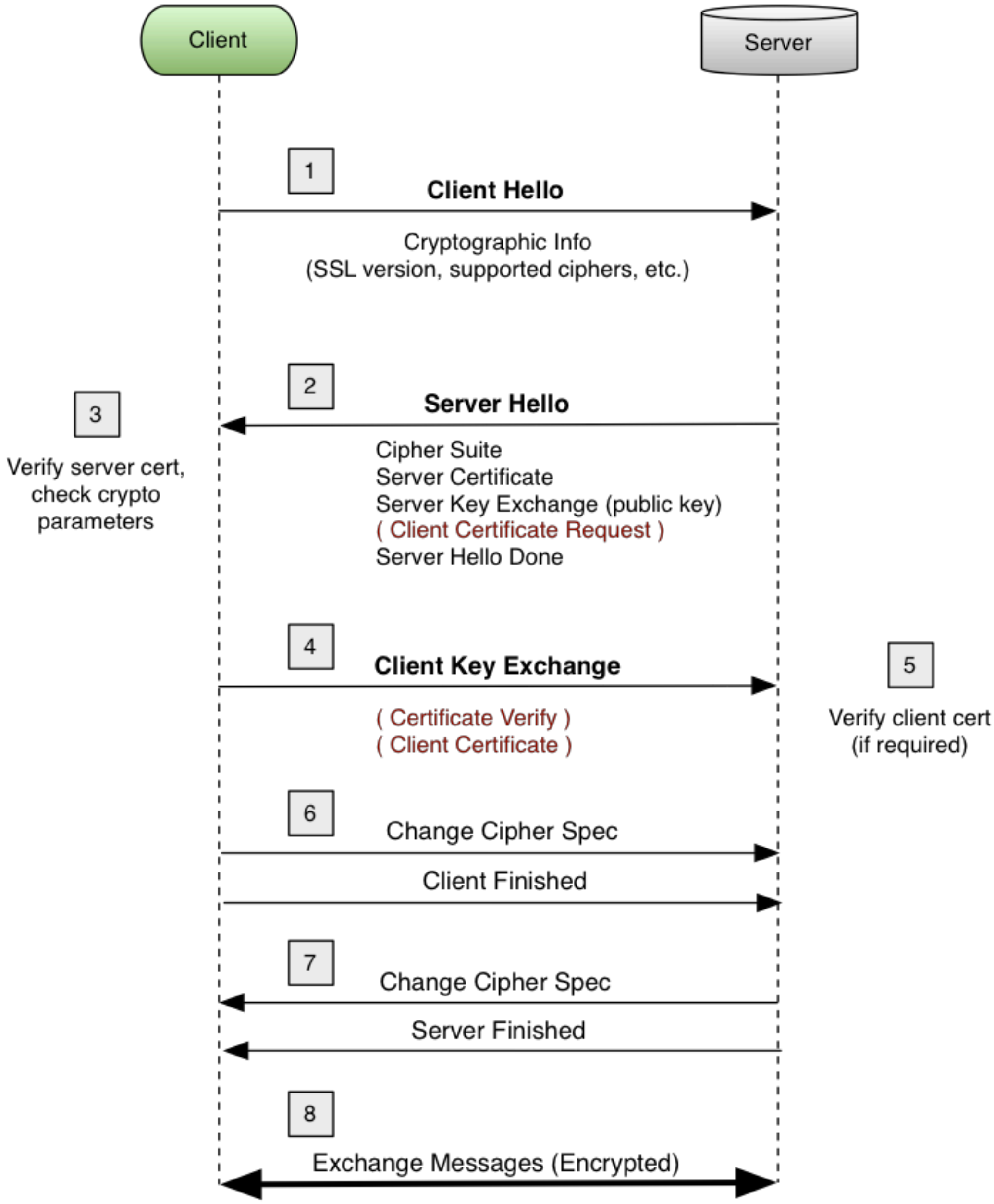
A general diagram of how the SSL protocol fits into existing protocols can be seen in **Figure 1**. SSL sits in between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the transport medium. Application protocols are layered on top of SSL (such as HTTP, FTP, and SMTP).



(Figure 1: SSL Protocol Diagram)

A.2 SSL Handshake

The SSL handshake involves several steps, some of which are optional depending on what options the SSL client and server have been configured with. Below, in **Figure 2**, you will find a simplified diagram of the SSL handshake process.



(Figure 2: SSL Handshake Diagram)

A.3 Differences between SSL and TLS Protocol Versions

SSL (Secure Socket Layer) and TLS (Transport Security Layer) are both cryptographic protocols which provide secure communication over networks. These two protocols (and the several version of each) are in widespread use today in applications ranging from web browsing to e-mail to instant messaging and VoIP. Each protocol, and the underlying versions of each, are slightly different from the other.

Below you will find both an explanation of and the major differences between the different SSL and TLS protocol versions. For specific details about each protocol, please reference the RFC specification mentioned.

SSL 3.0
This protocol was released in 1996 but began with the creation of SSL 1.0 developed by Netscape. Version 1.0 wasn't released, and version 2.0 had a number of security flaws, thus leading to the release of SSL 3.0. Some major improvements of SSL 3.0 over SSL 2.0 are:

- Separation of the transport of data from the message layer
- Use of a full 128 bits of keying material even when using the Export cipher
- Ability of the client and server to send chains of certificates, thus allowing organizations to use certificate hierarchy which is more than two certificates deep.
- Implementing a generalized key exchange protocol, allowing Diffie-Hellman and Fortezza key exchanges as well as non-RSA certificates.
- Allowing for record compression and decompression
- Ability to fall back to SSL 2.0 when a 2.0 client is encountered

TLS 1.0
This protocol was first defined in RFC 2246 in January of 1999. This was an upgrade from SSL 3.0 and the differences were not dramatic, but they are significant enough that SSL 3.0 and TLS 1.0 don't interoperate. Some of the major differences between SSL 3.0 and TLS 1.0 are:

- Key derivation functions are different
- MACs are different - SSL 3.0 uses a modification of an early HMAC while TLS 1.0 uses HMAC.
- The Finished messages are different
- TLS has more alerts
- TLS requires DSS/DH support

TLS 1.1
This protocol was defined in RFC 4346 in April of 2006, and is an update to TLS 1.0. The major changes are:

- The Implicit Initialization Vector (IV) is replaced with an explicit IV to protect against Cipher block chaining (CBC) attacks.

- Handling of padded errors is changed to use the bad_record_mac alert rather than the decryption_failed alert to protect against CBC attacks.
- IANA registries are defined for protocol parameters
- Premature closes no longer cause a session to be non-resumable.

TLS 1.2

This protocol was defined in RFC 5246 in August of 2008. Based on TLS 1.1, TLS 1.2 contains improved flexibility. The major differences include:

- The MD5/SHA-1 combination in the pseudorandom function (PRF) was replaced with cipher-suite-specified PRFs.
- The MD5/SHA-1 combination in the digitally-signed element was replaced with a single hash. Signed elements include a field explicitly specifying the hash algorithm used.
- There was substantial cleanup to the client's and server's ability to specify which hash and signature algorithms they will accept.
- Addition of support for authenticated encryption with additional data modes.
- TLS Extensions definition and AES Cipher Suites were merged in.
- Tighter checking of EncryptedPreMasterSecret version numbers.
- Many of the requirements were tightened
- Verify_data length depends on the cipher suite
- Description of Bleichenbacher/Dlima attack defenses cleaned up.

TLS 1.3

This protocol was defined in an Internet Draft in April of 2017. TLS 1.3 contains improved security and speed. The major differences include:

- The list of supported symmetric algorithms has been pruned of all legacy algorithms. The remaining algorithms all use Authenticated Encryption with Associated Data (AEAD) algorithms.
- A zero-RTT (0-RTT) mode was added, saving a round-trip at connection setup for some application data at the cost of certain security properties.
- All handshake messages after the ServerHello are now encrypted.

- Key derivation functions have been re-designed, with the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) being used as a primitive.
- The handshake state machine has been restructured to be more consistent and remove superfluous messages.
- ECC is now in the base spec and includes new signature algorithms. Point format negotiation has been removed in favor of single point format for each curve.
- Compression, custom DHE groups, and DSA have been removed, RSA padding now uses PSS.
- TLS 1.2 version negotiation verification mechanism was deprecated in favor of a version list in an extension.
- Session resumption with and without server-side state and the PSK-based ciphersuites of earlier versions of TLS have been replaced by a single new PSK exchange.

Appendix B: RFCs, Specifications, and Reference

B.1 Protocols

SSL v3.0	http://tools.ietf.org/id/draft-ietf-tls-ssl-version3-00.txt
TLS v1.0	http://www.ietf.org/rfc/rfc2246.txt
TLS v1.1	http://www.ietf.org/rfc/rfc4346.txt
TLS v1.2	http://www.ietf.org/rfc/rfc5246.txt
TLS v1.3	https://tools.ietf.org/html/draft-ietf-tls-tls13-20
DTLS	http://tools.ietf.org/html/rfc4347 http://crypto.stanford.edu/~nagendra/papers/dtls.pdf
IPv4	http://en.wikipedia.org/wiki/IPv4
IPv6	http://en.wikipedia.org/wiki/IPv6

B.2 Stream Ciphers

Stream Cipher	http://en.wikipedia.org/wiki/Stream_cipher
HC-128	http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf
RABBIT	http://cr.yp.to/streamciphers/rabbit/desc.pdf
RC4 / ARC4	http://tools.ietf.org/id/draft-kaukonen-cipher-arcfour-03.txt http://en.wikipedia.org/wiki/Rc4

B.3 Block Ciphers

Block Cipher	http://en.wikipedia.org/wiki/Block_cipher
AES	http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
AES-GCM	http://www.csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf
AES-NI	Intel Software Network
DES/3DES	http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf http://en.wikipedia.org/wiki/Data_Encryption_Standard

B.4 Hashing Functions

SHA	http://www.itl.nist.gov/fipspubs/fip180-1.htm http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf http://en.wikipedia.org/wiki/SHA_hash_functions
MD4	http://tools.ietf.org/html/rfc1320
MD5	http://tools.ietf.org/html/rfc1321
RIPEMD-160	http://homes.esat.kuleuven.be/~bosselae/ripemd160.html

B.5 Public Key Cryptography

Diffie-Hellman	http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange
RSA	http://people.csail.mit.edu/rivest/Rsapaper.pdf http://en.wikipedia.org/wiki/RSA
DSA/DSS	http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf
DSA/DSS	http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf
NTRU	http://securityinnovation.com/cryptolab/
X.509	http://www.ietf.org/rfc/rfc3279.txt
ASN.1	http://luca.ntop.org/Teaching/Appunti/asn1.html http://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One
PSK	http://tools.ietf.org/html/rfc4279

B.6 Other

PKCS#5, PBKDF1, PBKDF2	http://tools.ietf.org/html/rfc2898
PKCS#8	http://tools.ietf.org/html/rfc5208
PKCS#12	http://www.rsa.com/rsalabs/node.asp?id=2138

Appendix C: Error Codes

C.1 wolfSSL Error Codes

wolfSSL (formerly CyaSSL) error codes can be found in **wolfssl/ssl.h**. For detailed descriptions of the following errors, see the OpenSSL man page for **SSL_get_error** (man SSL_get_error).

Error Code Enum	Error Code	Error Description
SSL_ERROR_WANT_READ	2	
SSL_ERROR_WANT_WRITE	3	
SSL_ERROR_WANT_CONNECT	7	
SSL_ERROR_WANT_ACCEPT	8	
SSL_ERROR_SYSCALL	5	
SSL_ERROR_WANT_X509_LOOKUP	83	
SSL_ERROR_ZERO_RETURN	6	
SSL_ERROR_SSL	85	

Additional wolfSSL error codes can be found in **wolfssl/error-ssl.h**

Error Code Enum	Error Code	Error Description
PREFIX_ERROR	-302	bad index to key rounds
MEMORY_ERROR	-303	out of memory
VERIFY_FINISHED_ERROR	-304	verify problem on finished
VERIFY_MAC_ERROR	-305	verify mac problem
PARSE_ERROR	-306	parse error on header
UNKNOWN_HANDSHAKE_TYPE	-307	weird handshake type

SOCKET_ERROR_E	-308	error state on socket
SOCKET_NODATA	-309	expected data, not there
INCOMPLETE_DATA	-310	don't have enough data to complete task
UNKNOWN_RECORD_TYPE	-311	unknown type in record hdr
DECRYPT_ERROR	-312	error during decryption
FATAL_ERROR	-313	revcd alert fatal error
ENCRYPT_ERROR	-314	error during encryption
FREAD_ERROR	-315	fread problem
NO_PEER_KEY	-316	need peer's key
NO_PRIVATE_KEY	-317	need the private key
RSA_PRIVATE_ERROR	-318	error during rsa priv op
NO_DH_PARAMS	-319	server missing DH params
BUILD_MSG_ERROR	-320	build message failure
BAD_HELLO	-321	client hello malformed
DOMAIN_NAME_MISMATCH	-322	peer subject name mismatch
WANT_READ	-323	want read, call again
NOT_READY_ERROR	-324	handshake layer not ready
PMS_VERSION_ERROR	-325	pre m secret version error
VERSION_ERROR	-326	record layer version error
WANT_WRITE	-327	want write, call again
BUFFER_ERROR	-328	malformed buffer input
VERIFY_CERT_ERROR	-329	verify cert error
VERIFY_SIGN_ERROR	-330	verify sign error
CLIENT_ID_ERROR	-331	psk client identity error
SERVER_HINT_ERROR	-332	psk server hint error
PSK_KEY_ERROR	-333	psk key error
ZLIB_INIT_ERROR	-334	zlib init error
ZLIB_COMPRESS_ERROR	-335	zlib compression error
ZLIB_DECOMPRESS_ERROR	-336	zlib decompression error
GETTIME_ERROR	-337	gettimeofday failed ???
GETITIMER_ERROR	-338	getitimer failed ???
SIGACT_ERROR	-339	sigaction failed ???

SETITIMER_ERROR	-340	setitimer failed ???
LENGTH_ERROR	-341	record layer length error
PEER_KEY_ERROR	-342	cant decode peer key
ZERO_RETURN	-343	peer sent close notify
SIDE_ERROR	-344	wrong client/server type
NO_PEER_CERT	-345	peer didn't send key
NTRU_KEY_ERROR	-346	NTRU key error
NTRU_DRBG_ERROR	-347	NTRU drbg error
NTRU_ENCRYPT_ERROR	-348	NTRU encrypt error
NTRU_DECRYPT_ERROR	-349	NTRU decrypt error
ECC_CURVETYPE_ERROR	-350	Bad ECC Curve Type
ECC_CURVE_ERROR	-351	Bad ECC Curve
ECC_PEERKEY_ERROR	-352	Bad Peer ECC Key
ECC_MAKEKEY_ERROR	-353	Bad Make ECC Key
ECC_EXPORT_ERROR	-354	Bad ECC Export Key
ECC_SHARED_ERROR	-355	Bad ECC Shared Secret
NOT_CA_ERROR	-357	Not CA cert error
BAD_PATH_ERROR	-358	Bad path for opendir
BAD_CERT_MANAGER_ERROR	-359	Bad Cert Manager
OCSP_CERT_REVOKED	-360	OCSP Certificate revoked
CRL_CERT_REVOKED	-361	CRL Certificate revoked
CRL_MISSING	-362	CRL Not loaded
MONITOR_RUNNING_E	-363	CRL Monitor already running
THREAD_CREATE_E	-364	Thread Create Error
OCSP_NEED_URL	-365	OCSP need an URL for lookup
OCSP_CERT_UNKNOWN	-366	OCSP responder doesn’t know
OCSP_LOOKUP_FAIL	-367	OCSP lookup not successful
MAX_CHAIN_ERROR	-368	max chain depth exceeded
COOKIE_ERROR	-369	dtls cookie error
SEQUENCE_ERROR	-370	dtls sequence error
SUITES_ERROR	-371	suites pointer error
SSL_NO_PEM_HEADER	-372	no PEM header found

OUT_OF_ORDER_E	-373	out of order message
BAD_KEA_TYPE_E	-374	bad KEA type found
SANITY_CIPHER_E	-375	sanity check on cipher error
RECV_OVERFLOW_E	-376	RXCB returned more than rqed
GEN_COOKIE_E	-377	Generate Cookie Error
NO_PEER_VERIFY	-378	Need peer cert verify Error
FWRITE_ERROR	-379	fwrite problem
CACHE_MATCH_ERROR	-380	cache hrd match error
UNKNOWN_SNI_HOST_NAME_E	-381	Unrecognized host name Error
UNKNOWN_MAX_FRAG_LEN_E	-382	Unrecognized max frag len Error
KEYUSE_SIGNATURE_E	-383	KeyUse digSignature error
KEYUSE_ENCIPHER_E	-385	KeyUse KeyEncipher error
EXTKEYUSE_AUTH_E	-386	ExtKeyUse server|client_auth
SEND_OOB_READ_E	-387	Send Cb out of bounds read
UNSUPPORTED_SUITE	-390	unsupported cipher suite
MATCH_SUITE_ERROR	-391	can't match cipher suite

C.2 wolfCrypt Error Codes

wolfCrypt error codes can be found in **wolfssl/wolfcrypt/error.h**.

Error Code Enum	**Error Code**	**Error Description**
OPEN_RAN_E	-101	opening random device error
READ_RAN_E	-102	reading random device error
WINCRYPT_E	-103	windows crypt init error
CRYPTGEN_E	-104	windows crypt generation error
RAN_BLOCK_E	-105	reading random device would block
BAD_MUTEX_E	-106	Bad mutex operation
MP_INIT_E	-110	mp_init error state
MP_READ_E	-111	mp_read error state
MP_EXPTMOD_E	-112	mp_exptmod error state

MP_TO_E	-113	mp_to_xxx error state, can't convert
MP_SUB_E	-114	mp_sub error state, can't subtract
MP_ADD_E	-115	mp_add error state, can't add
MP_MUL_E	-116	mp_mul error state, can't multiply
MP_MULMOD_E	-117	mp_mulmod error state, can't multiply mod
MP_MOD_E	-118	mp_mod error state, can't mod
MP_INVMOD_E	-119	mp_invmod error state, can't inv mod
MP_CMP_E	-120	mp_cmp error state
MP_ZERO_E	-121	got a mp zero result, not expected
MEMORY_E	-125	out of memory error
RSA_WRONG_TYPE_E	-130	RSA wrong block type for RSA function
RSA_BUFFER_E	-131	RSA buffer error, output too small or input too large
BUFFER_E	-132	output buffer too small or input too large
ALGO_ID_E	-133	setting algo id error
PUBLIC_KEY_E	-134	setting public key error
DATE_E	-135	setting date validity error
SUBJECT_E	-136	setting subject name error
ISSUER_E	-137	setting issuer name error
CA_TRUE_E	-138	setting CA basic constraint true error
EXTENSIONS_E	-139	setting extensions error
ASN_PARSE_E	-140	ASN parsing error, invalid input
ASN_VERSION_E	-141	ASN version error, invalid number
ASN_GETINT_E	-142	ASN get big int error, invalid data
ASN_RSA_KEY_E	-143	ASN key init error, invalid input
ASN_OBJECT_ID_E	-144	ASN object id error, invalid id
ASN_TAG_NULL_E	-145	ASN tag error, not null
ASN_EXPECT_0_E	-146	ASN expect error, not zero
ASN_BITSTR_E	-147	ASN bit string error, wrong id
ASN_UNKNOWN_OID_E	-148	ASN oid error, unknown sum id
ASN_DATE_SZ_E	-149	ASN date error, bad size
ASN_BEFORE_DATE_E	-150	ASN date error, current date before
ASN_AFTER_DATE_E	-151	ASN date error, current date after

ASN_SIG_OID_E	-152	ASN signature error, mismatched oid
ASN_TIME_E	-153	ASN time error, unknown time type
ASN_INPUT_E	-154	ASN input error, not enough data
ASN_SIG_CONFIRM_E	-155	ASN sig error, confirm failure
ASN_SIG_HASH_E	-156	ASN sig error, unsupported hash type
ASN_SIG_KEY_E	-157	ASN sig error, unsupported key type
ASN_DH_KEY_E	-158	ASN key init error, invalid input
ASN_NTRU_KEY_E	-159	ASN ntru key decode error, invalid input
ASN_CRIT_EXT_E	-160	ASN unsupported critical extension
ECC_BAD_ARG_E	-170	ECC input argument of wrong type
ASN_ECC_KEY_E	-171	ASN ECC bad input
ECC_CURVE_OID_E	-172	Unsupported ECC OID curve type
BAD_FUNC_ARG	-173	Bad function argument provided
NOT_COMPILED_IN	-174	Feature not compiled in
UNICODE_SIZE_E	-175	Unicode password too big
NO_PASSWORD	-176	no password provided by user
ALT_NAME_E	-177	alt name size problem, too big
AES_GCM_AUTH_E	-180	AES-GCM Authentication check failure
AES_CCM_AUTH_E	-181	AES-CCM Authentication check failure
CAVIUM_INIT_E	-182	Cavium Init type error
COMPRESS_INIT_E	-183	Compress init error
COMPRESS_E	-184	Compress error
DECOMPRESS_INIT_E	-185	DeCompress init error
DECOMPRESS_E	-186	DeCompress error
BAD_ALIGN_E	-187	Bad alignment for operation, no alloc
ASN_NO_SIGNER_E	-188	ASN sig error, no CA signer to verify certificate
ASN_CRL_CONFIRM_E	-189	ASN CRL no signer to confirm failure
ASN_CRL_NO_SIGNER_E	-190	ASN CRL no signer to confirm failure
ASN_OCSP_CONFIRM_E	-191	ASN OCSP signature confirm failure
BAD_ENC_STATE_E	-192	Bad ecc enc state operation
BAD_PADDING_E	-193	Bad padding, msg not correct length
REQ_ATTRIBUTE_E	-194	Setting cert request attributes error

PKCS7_OID_E	-195	PKCS#7, mismatched OID error
PKCS7_RECIP_E	-196	PKCS#7, recipient error
FIPS_NOT_ALLOWED_E	-197	FIPS not allowed error
ASN_NAME_INVALID_E	-198	ASN name constraint error
RNG_FAILURE_E	-199	RNG Failed, Reinitialize
HMAC_MIN_KEYLEN_E	-200	FIPS Mode HMAC Minimum Key Length error
RSA_PAD_E	-201	RSA Padding Error
LENGTH_ONLY_E	-202	Returning output length only
IN_CORE_FIPS_E	-203	In Core Integrity check failure
AES_KAT_FIPS_E	-204	AES KAT failure
DES3_KAT_FIPS_E	-205	DES3 KAT failure
HMAC_KAT_FIPS_E	-206	HMAC KAT failure
RSA_KAT_FIPS_E	-207	RSA KAT failure
DRBG_KAT_FIPS_E	-208	HASH DRBG KAT failure
DRBG_CONT_FIPS_E	-209	HASH DRBG Continuous test failure
AESGCM_KAT_FIPS_E	-210	AESGCM KAT failure
THREAD_STORE_KEY_E	-211	Thread local storage key create failure
THREAD_STORE_SET_E	-212	Thread local storage key set failure
MAC_CMP_FAILED_E	-213	MAC comparison failed
IS_POINT_E	-214	ECC is point on curve failed
ECC_INF_E	-215	ECC point infinity error
ECC_PRIV_KEY_E	-216	ECC private key not valid error
SRP_CALL_ORDER_E	-217	SRP function called in the wrong order
SRP_VERIFY_E	-218	SRP proof verification failed
SRP_BAD_KEY_E	-219	SRP bad ephemeral values
ASN_NO_SKID	-220	ASN no Subject Key Identifier found
ASN_NO_AKID	-221	ASN no Authority Key Identifier found
ASN_NO_KEYUSAGE	-223	ASN no Key Usage found
SKID_E	-224	Setting Subject Key Identifier error
AKID_E	-225	Setting Authority Key Identifier error
KEYUSAGE_E	-226	Bad Key Usage value
CERTPOLICIES_E	-227	Setting Certificate Policies error

WC_INIT_E	-228	wolfCrypt failed to initialize
SIG_VERIFY_E	-229	wolfCrypt signature verify error
BAD_COND_E	-230	Bad condition variable operation
SIG_TYPE_E	-231	Signature Type not enabled/available
HASH_TYPE_E	-232	Hash Type not enabled/available
MIN_CODE_E	-300	errors -101 - -299

C.3 Common Error Codes and their Solution

There are several error codes that commonly happen when getting an application up and running with wolfSSL.

ASN_NO_SIGNER_E (-188)
This error occurs when using a certificate and the signing CA certificate was not loaded. This can be seen using the wolfSSL example server or client against another client or server, for example connecting to Google using the wolfSSL example client:

```
$ ./examples/client/client -g -h www.google.com -p 443
```

This fails with error -188 because Google’s CA certificate wasn’t loaded with the “`-A`” command line option.

WANT_READ (-323)
The WANT_READ error happens often when using non-blocking sockets, and isn’t actually an error when using non-blocking sockets, but it is passed up to the caller as an error. When a call to receive data from the I/O callback would block as there isn’t data currently available to receive, the I/O callback returns WANT_READ. The caller should wait and try receiving again later. This is usually seen from calls to wolfSSL_read(), wolfSSL_negotiate(), wolfSSL_accept(), and wolfSSL_connect(). The example client and server will indicate the WANT_READ incidents when debugging is enabled.