wolfSSL User Manual

March 18, 2015
Version 3.3.0

Table of Contents

Chapter 1: Introduction

This manual is written as a technical guide to the wolfSSL, formerly CyaSSL, embedded SSL 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, 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 (formerly CyaSSL) 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 a 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/yaSSL/download/downloadForm.php

After downloading the ZIP file, unzip the file using the "**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. (For example, Unix files use line feeds (LFs) for end-of-line (EOL) and have no end-of-file (EOF) marker; Apple Operating Systems use carriage returns (CRs) for EOLs; and most PC operating systems use CR+LF for EOLs and control-Z for EOF. In addition, IBM mainframes and the Michigan Terminal System use EBCDIC rather than the more common ASCII character set, and NT supports Unicode.)"

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:

```
./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 source 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

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 propery 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" click the drop down arrow on the far right and 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 the project then Rebuild the project and 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.

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 sub-sections, below. For more verbose descriptions of many options, please see section 2.5.1, “Build Option Notes”.

2.4.1 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.

2.4.2 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 so it is off by default.

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. It is off by default.

HAVE_AESCCM enables AES-CCM support.

HAVE_AESGCM enables AES-GCM 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_ECC enables Elliptical Curve Cryptography (ECC) support.

HAVE_LIBZ is an extension that can allow for compression of data over the connection. It is off by default and 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.

2.4.3 Customizing or Porting wolfSSL

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.

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.

2.4.4 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.

2.4.5 Increasing Performance

WOLFSSL_AESNI enables use of AES accelerated operations which are built into some Intel chipsets. When using this define, the aes_asm.s 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.

2.4.6 Stack or Chip Specific Defines

wolfSSL can be built for a variety of platforms and TCP/IP stacks. 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 for Micrium's µC/OS (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.

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 define 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_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.

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-debug	Disabled	Enable wolfSSL debugging support
--enable-singlethreaded	Disabled	Enable single threaded mode, no multi thread protections
--enable-dtls	Disabled	Enable wolfSSL DTLS support
--enable-opensslextra	Disabled	Enable extra OpenSSL API compatibility, increases the size
--enable-ipv6	Disabled	Enable testing of IPv6, wolfSSL proper is IP neutral
--enable-bump	Disabled	Enable SSL Bump build
--enable-leanpsk	Disabled	Enable Lean PSK build
--enable-bigcache	Disabled	Enable a big session cache
--enable-hugecache	Disabled	Enable a huge session cache
--enable-smallcache	Disabled	Enable small session cache
--enable-savesession	Disabled	Enable persistent session cache
--enable-savecert	Disabled	Enable persistent cert cache
--enable-atomicuser	Disabled	Enable Atomic User Record Layer

--enable-pkcallbacks	Disabled	Enable Public Key Callbacks
--enable-sniffer	Disabled	Enable wolfSSL sniffer support
--enable-aesgcm	Disabled	Enable AES-GCM support
--enable-aesccm	Disabled	Enable AES-CCM support
--enable-aesni	Disabled	Enable wolfSSL Intel AES-NI support
--enable-poly1305	Disabled	Enable Poly1305
--enable-camellia	Disabled	Enable Camellia support
--enable-md2	Disabled	Enable MD2 support
--enable-nullcipher	Disabled	Enable wolfSSL NULL cipher support
--enable-ripemd	Disabled	Enable wolfSSL RIPEMD-160 support
--enable-blake2	Disabled	Enable wolfSSL BLAKE2 support
--enable-sha512	Disabled	Enable wolfSSL SHA-512 support
--enable-sessioncerts	Disabled	Enable session cert storing
--enable-keygen	Disabled	Enable key generation
--enable-certgen	Disabled	Enable cert generation
--enable-certreq	Disabled	Enable cert request generation
--enable-sep	Disabled	Enable SEP extensions
--enable-hkdf	Disabled	Enable HKDF (HMAC-KDF)
--enable-dsa	Disabled	Enable DSA
--enable-ecc	Disabled	Enable ECC
--enable-fpecc	Disabled	Enable Fixed Point cache ECC
--enable-eccencrypt	Disabled	Enable ECC encrypt

--enable-psk	Disabled	Enable PSK (Pre Shared Keys)
--enable-errorstrings	Enabled	Enable error strings table
--enable-oldtls	Enabled	Enable old TLS version < 1.2
--enable-stacksize	Disabled	Enable stack size info on examples
--enable-memory	Enabled	Enable memory callbacks
--enable-rsa	Enabled	Enable RSA
--enable-dh	Disabled	Enable DH
--enable-asn	Enabled	Enable ASN
--enable-aes	Enabled	Enable AES
--enable-coding	Enabled	Enable Coding base 16/64
--enable-des3	Enabled	Enable DES3
--enable-arc4	Enabled	Enabled ARC4
--enable-md5	Enabled	Enable MD5
--enable-sha	Enabled	Enable SHA
--enable-md4	Disabled	Enable MD4
--enable-pwdbased	Disabled	Enable PWDBASED
--enable-hc128	Disabled	Enable streaming cipher HC-128
--enable-rabbit	Disabled	Enable streaming cipher RABBIT
--enable-chacha	Disabled	Enable ChaCha20
--enable-fips	Disabled	Enable FIPS 140-2
--enable-hashdrbg	Disabled	Enable Hash DRBG support
--enable-filesystem	Enabled	Enable Filesystem support
--enable-inline	Enabled	Enable inline functions

--enable-ocsp	Disabled	Enable OCSP
--enable-crl	Disabled	Enable CRL
--enable-crl-monitor	Disabled	Enable CRL Monitor
--enable-sni	Disabled	Enable SNI
--enable-maxfragment	Disabled	Enable Maximum Fragment Length
--enable-truncatedhmac	Disabled	Enable Truncated HMAC
--enable-renegotiation-indication	Disabled	Enable Renegotiation Indication
--enable-supportedcurves	Disabled	Enable Supported Elliptic Curves
--enable-tlsx	Disabled	Enable all TLS extensions
--enable-pkcs7	Disabled	Enable PKCS#7 support
--enable-scep	Disabled	Enable wolfSCEP
--enable-smallstack	Disabled	Enable Small Stack Usage
--enable-valgrind	Disabled	Enable valgrind for unit tests
--enable-testcert	Disabled	Enable Test Cert
--enable-iopool	Disabled	Enable I/O Pool example
--enable-fastmath	Enabled on x86_64	Enable fast math ops
--enable-fasthugemath	Disabled	Enable fast math + huge code
--enable-examples	Enabled	Enable examples
--enable-mcapi	Disabled	Enable Microchip API
--enable-jobserver [=no/yes/#]	yes	Enable up to # make jobs yes: enable one more than CPU count
--disable-shared	Disabled	Disable the building of a shared wolfSSL library
--disable-static	Disabled	Disable the building of a static wolfSSL library

--with-ntru=PATH	Disabled	Path to NTRU install (default /usr/)
--with-libz=PATH	Disabled	Optionally include libz for compression
--with-cavium=PATH	Disabled	Path to cavium/software dir

2.5.1 Build Option Notes

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.

Single Threaded - 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.

OpenSSL Extra - 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.

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 8byte 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 steaming 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.

noInline - enabling this option disables function inlining in wolfSSL.

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.

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.

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.

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
```

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.

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!
MD4      test passed!
SHA      test passed!
SHA-256  test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA256 test passed!
ARC4     test passed!
DES      test passed!
DES3     test passed!
AES      test passed!
RANDOM   test passed!
RSA      test passed!
DH       test passed!
DSA      test passed!
PWDBASED test passed!
OPENSSL  test passed!
peer's cert info:
```

```
 issuer :
/C=US/ST=Oregon/L=Portland/O=wolfSSL/OU=Programming/CN=www.wolfs
sl.com/emailAddress=info@yassl.com
 subject:
/C=US/ST=Oregon/L=Portland/O=wolfSSL/OU=Programming/CN=www.wolfs
sl.com/emailAddress=info@yassl.com
 serial number:87:4a:75:be:91:66:d8:3d
SSL version is TLSv1.2
SSL cipher suite is TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
peer's cert info:
 issuer :
/C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.yassl
.com/emailAddress=info@yassl.com
 subject:
/C=US/ST=Montana/L=Bozeman/O=wolfSSL/OU=Support/CN=www.wolfssl.c
om/emailAddress=info@wolfssl.com
 serial number:02
SSL version is TLSv1.2
SSL cipher suite is TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
6cd8940c5e7229f9357cc15b202b593befbbc8ea  input
6cd8940c5e7229f9357cc15b202b593befbbc8ea  output

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
client 3.4.6 NOTE: All files relative to wolfSSL home dir
-?          Help, print this usage
-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 [0-3], SSLv3(0) - TLS1.2(3)), default 3
-l <str>    Cipher list
-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
-b <num>    Benchmark <num> connections and print stats
-s          Use pre Shared keys
-t          Track wolfSSL memory use
-d          Disable peer checks
-D          Override Date Errors example
-g          Send server HTTP GET
-u          Use UDP DTLS, add -v 2 for DTLSv1 (default), -v 3
for DTLSv1.2
-m          Match domain name in cert
-N          Use Non-blocking sockets
-r          Resume session
-w          Wait for bidirectional shutdown
-f          Fewer packets/group messages
-x          Disable client cert/key loading
```

To test against secure gmail try the following. This is using wolfSSL compiled with the --enable-opensslextra build option:

```
./examples/client/client -h gmail.google.com -p 443 -d -g
peer's cert info:
 issuer : /C=US/O=Google Inc/CN=Google Internet Authority
 subject: /C=US/ST=California/L=Mountain View/O=Google
Inc/CN=*.google.com
 altname = *.googleapis.cn
 altname = *.gstatic.com
 altname = g.co
 altname = goo.gl
 altname = *.cloud.google.com
```

```
 altname = google-analytics.com
 altname = *.google-analytics.com
 altname = urchin.com
 altname = *.urchin.com
 altname = *.url.google.com
 altname = googlecommerce.com
 altname = *.googlecommerce.com
 altname = android.com
 altname = *.android.com
 altname = *.google.com.tr
 altname = *.google.com.vn
 altname = *.google.com.co
 altname = *.google.com.ar
 altname = *.google.com.mx
 altname = *.google.hu
 altname = *.google.co.jp
 altname = *.google.com.au
 altname = *.google.nl
 altname = *.google.pl
 altname = *.google.cl
 altname = *.google.de
 altname = *.google.it
 altname = *.google.pt
 altname = *.google.fr
 altname = *.google.ca
 altname = *.google.co.uk
 altname = *.google.es
 altname = *.google.co.in
 altname = *.google.com.br
 altname = *.ytimg.com
 altname = youtu.be
 altname = *.youtube-nocookie.com
 altname = youtube.com
 altname = *.youtube.com
 altname = google.com
 altname = *.google.com
 serial number:40:98:f6:53:00:00:00:00:68:b6
SSL version is TLSv1.2
SSL cipher suite is SSL_RSA_WITH_RC4_128_SHA
SSL connect ok, sending GET...
Server response: HTTP/1.0 302 Found
```

```
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Location: http://www.google.com
Content-Length: 218
Date: Mon, 01 Oct 2012 21:17:18 GMT
Server: GFE/2.0
```

This tells the client to connect to gmail.google.com on the HTTPS port of 443 and sends a generic GET. 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 **yasslIP** and **yasslPort** 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
-v 1  =  TLS 1.0
-v 2  =  TLS 1.1
-v 3  =  TLS 1.2
```

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
```

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 3.4.6 NOTE: All files relative to wolfSSL home dir
-?          Help, print this usage
-p <num>    Port to listen on, not 0, default 11111
-v <num>    SSL version [0-3], SSLv3(0) - TLS1.2(3)), default 3
-l <str>    Cipher list
-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
-d          Disable client cert check
-b          Bind to any interface instead of localhost only
```

```
-s          Use pre Shared keys
-t          Track wolfSSL memory use
-u          Use UDP DTLS, add -v 2 for DTLSv1 (default), -v 3
for DTLSv1.2
-f          Fewer packets/group messages
-r          Create server ready file, for external monitor
-N          Use Non-blocking sockets
-S <str>    Use Host Name Indication
-w          Wait for bidirectional shutdown
```

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 echos 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 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):

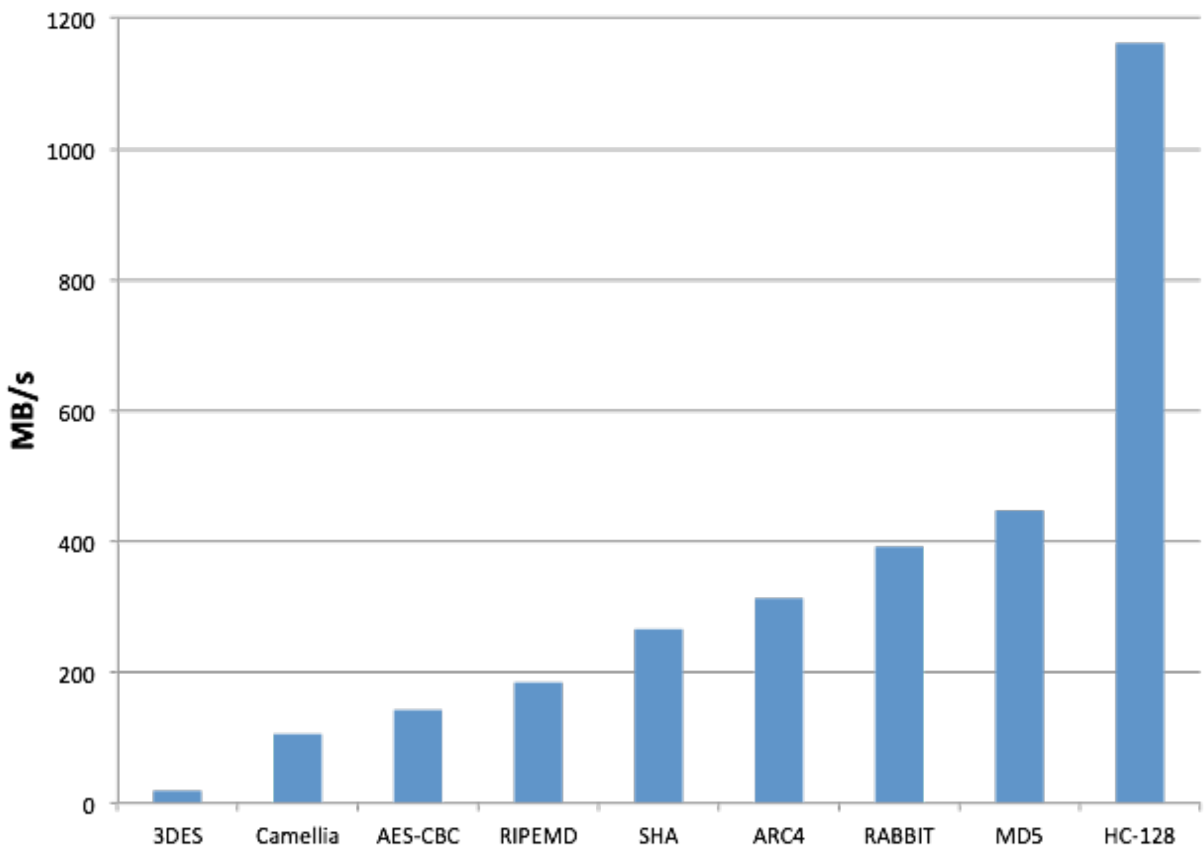
```
./wolfcrypt/benchmark/benchmark
AES      50 megs took 0.274 seconds,  182.588 MB/s Cycles per byte =  11.99
AES-GCM  50 megs took 0.824 seconds,   60.694 MB/s Cycles per byte =  36.06
AES-CCM  50 megs took 0.517 seconds,   96.697 MB/s Cycles per byte =  22.63
Camellia 50 megs took 0.367 seconds,  136.311 MB/s Cycles per byte =  16.05
HC128    50 megs took 0.030 seconds, 1651.847 MB/s Cycles per byte =   1.32
RABBIT   50 megs took 0.110 seconds,  452.555 MB/s Cycles per byte =   4.84
CHACHA   50 megs took 0.136 seconds,  366.617 MB/s Cycles per byte =   5.97
CHA-POLY 50 megs took 0.173 seconds,  288.301 MB/s Cycles per byte =   7.59
3DES     50 megs took 1.858 seconds,   26.907 MB/s Cycles per byte =  81.33

MD5      50 megs took 0.114 seconds,  440.273 MB/s Cycles per byte =   4.97
POLY1305 50 megs took 0.043 seconds, 1153.562 MB/s Cycles per byte =   1.90
SHA      50 megs took 0.103 seconds,  484.172 MB/s Cycles per byte =   4.52
SHA-256  50 megs took 0.240 seconds,  208.581 MB/s Cycles per byte =  10.49
SHA-384  50 megs took 0.159 seconds,  313.674 MB/s Cycles per byte =   6.98
SHA-512  50 megs took 0.206 seconds,  242.518 MB/s Cycles per byte =   9.02
BLAKE2b  50 megs took 0.091 seconds,  548.120 MB/s Cycles per byte =   3.99

RSA 2048 encryption took  0.080 milliseconds, avg over 100 iterations
RSA 2048 decryption took  2.026 milliseconds, avg over 100 iterations
DH  2048 key generation   0.793 milliseconds, avg over 100 iterations
DH  2048 key agreement    0.763 milliseconds, avg over 100 iterations

ECC  256 key generation   0.426 milliseconds, avg over 100 iterations
EC-DHE   key agreement    0.421 milliseconds, avg over 100 iterations
EC-DSA   sign   time      0.451 milliseconds, avg over 100 iterations
EC-DSA   verify time      0.612 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: http://www.wolfssl.com/yaSSL/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 processors **native register size** (32 vs 64-bit) can make a big difference when doing 1000+ bit public key operations.

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

3. **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.

4. 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 CFLAGS=DTFM_NO_ASM
   ```

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

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

6. 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.

7. **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 except 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 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

The following table lists features included in the most recent release of wolfSSL.

wolfSSL Features (version 3.4.6)	**Benefits**
SSL version 3 and TLS versions 1, 1.1 and 1.2 (client and server)	Support for the most up to date standards with backwards compatibility
DTLS 1.0, 1.2 support (client and server)	Streaming Multimedia
Minimum footprint size of **20-100 kB**, depending on build options and operating environment	Small build size for use in resource constrained environments
Runtime memory usage between **1-36 kB** (depending on I/O buffer sizes, public key algorithm, and key size)	Minimal dynamic memory use, perfect for embedded systems or scalable enterprise servers.
OpenSSL Compatibility Layer	Standard API and ease of migration from

	OpenSSL
OCSP and CRL support	Used to confirm certificate validity
Multiple Hash Functions: MD2, MD4, MD5, SHA-1, SHA-2 (SHA-256, SHA-384, SHA-512), BLAKE2b, RIPEMD-160, Poly1305	
Block and Stream Ciphers: AES (CBC, CTR, GCM, CCM-8), Camellia, DES, 3DES, ARC4, RABBIT, HC-128, ChaCha20	4 block ciphers, 4 stream ciphers
Public Key Options: RSA, DSS, DH, EDH, NTRU	5 public key options
Password-based Key Derivation: HMAC, PBKDF2, PKCS #5	3 password-based key derivation options
ECC Support: ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-RSA	
RSA Key Generation	Fast run-time key generation support
Client Authentication Support	Provides ability to do mutual authentication between client and server, using certificates to verify clients.
PSK (Pre-Shared Keys) Support	Helpful for embedded devices lacking resources to do public key operations. Avoid RSA operations in limited environments.
Simple API	Easy to learn and use
zlib Compression Support	Highly configurable compression support
PEM and DER certificate support	No need to reconfigure certificates or keys
X.509 v3 Signed Certificate Generation	Generate your own certificates
Certificate Manager	Verify certs, check CRL outside of SSL usage

Intel AES-NI Support	Super fast chip-level AES encryption
STM32F2/F4 Hardware Crypto Support	Accelerate crypto on STM32 processors
Cavium NITROX Support	Accelerate crypto and SSL using Cavium NITROX.
Sniffer (SSL Inspection) Support	Decode SSL encrypted packets
Abstraction Layers / Callbacks C Standard Library, Custom I/O, Memory hooks, Logging callbacks, User Atomic Record Layer Processing, Public Key (RSA,ECC) callbacks	Providing more flexibility and portability to developers
IPv4 and IPv6 support	Compatible with current and upcoming protocols
PKCS #8 (PKCS #5, #12 formats)	Private Key Encryption
MySQL Integration	Wide distribution and testing
Supported Web Servers - yaSSLEWS, GoAhead, Mongoose, Lighttpd	Multiple lightweight embedded web server options. wolfSSL is also used in the yaSSL Embedded Web Server.

(Table 1: wolfSSL Features)

4.1.2 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.

4.2 Protocol Support

wolfSSL supports **SSL 3.0**, **TLS** (**1.0**, **1.1** and **1.2**), 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
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
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 the “Getting Started” Chapter. 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 a SSL 3.0 only server 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 number 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 TLS 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 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, the same area where wolfSSL is an ideal match for SSL/TLS needs.

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 an 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 short cut 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. 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 3.4.6)	
TLS_DHE_RSA_WITH_AES_256_CBC_SHA TLS_DHE_RSA_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_128_CBC_SHA TLS_PSK_WITH_NULL_SHA256 TLS_PSK_WITH_NULL_SHA SSL_RSA_WITH_RC4_128_SHA	

SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_3DES_EDE_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	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_CBC_MD5 TLS_RSA_WITH_HC_128_CBC_SHA TLS_RSA_WITH_RABBIT_CBC_SHA	wolfSSL extension - eSTREAM cipher suites
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	SHA-256 cipher suites

TLS_RSA_WITH_AES_128_CBC_SHA256 TLS_RSA_WITH_NULL_SHA256	
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	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_SHA256 TLS_RSA_WITH_AES_256_CCM_8_SHA384 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	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	ChaCha cipher suites

(Table 4: wolfSSL Cipher Suites)

4.3.3 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 performs about .11 times faster then 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.

4.3.3.1 What’s the Difference?

Have you ever wondered what the difference was between a block cipher and a stream cipher?

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.

4.3.4 Hashing Functions

wolfSSL supports several different hashing functions, including **MD2**, **MD4**, **MD5**, **SHA-1**, **SHA-2** (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.

4.3.5 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:

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 package 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 *<wolfssl_root>/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).

4.3.6 ECC Support

wolfSSL has support for Elliptic Curve Cryptography (ECC) including ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, 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 Table 4, above. ECC is disabled by default, 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. 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 ECDH-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.

4.3.7 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 #5**, **PKCS #8**, and PBKD from **PKCS #12**.

4.3.7.1 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 psudorandom 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 enable this functionality is to use the option `--enable-pwdbased`

A full example can be found in **wolfcrypt/src/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

4.3.7.2 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 non-encrypted 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

4.3.8 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:

```
SSL_CTX_set_cipher_list(ctx, “AES128-SHA”);
```

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

```
ctx = SSL_CTX_new(method);
SSL_CTX_set_cipher_list(ctx, “AES128-SHA”);
```

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 Intel 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 has added AES instructions at the chip level that perform the computational-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.

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 STM32F2

wolfSSL is able to use the STM32F2 hardware-based cryptography and random number generator through the STM32F2 Standard Peripheral Library.

For necessary defines, see the **WOLFSSL_STM32F2** define in settings.h. The WOLFSSL_STM32F2 define enables STM32F2 hardware crypto and RNG support by default. The defines for enabling these individually are **STM32F2_CRYPTO** (for hardware crypto support) and **STM32F2_RNG** (for hardware RNG support).

Documentation for the STM32F2 Standard Peripheral Library can be found in the following document:
http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/DM00023896.pdf

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.

4.5 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. There are five main sniffer functions which can be found in *sniffer.h*. They are listed below with a short description of each:

ssl_SetPrivateKey - Sets the private key for a specific server and port.
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.

To look at wolfSSL's sniffer support and see a complete example, please see the "**snifftest**" app in the "ssSniffer/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.

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 two ciphers with pre shared keys:

> **TLS_PSK_WITH_AES_256_CBC_SHA**
> **TLS_PSK_WITH_AES_128_CBC_SHA256**
> **TLS_PSK_WITH_AES_128_CBC_SHA**
> **TLS_PSK_WITH_NULL_SHA256**
> **TLS_PSK_WITH_NULL_SHA**
> **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**

These suites are automatically built into wolfSSL, though they 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.

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 |
                     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
```

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 setup 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 XSTRING_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 XSTRING_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 2 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, WinCE, Haiku, OpenWRT, iPhone (iOS), Android, Nintendo Wii and Gamecube through DevKitPro, QNX, MontaVista, NonStop, TRON/ITRON/µITRON, Micrium's µC/OS, FreeRTOS, SafeRTOS, Freescale MQX, Nucleus, TinyOS, HP/UX, TIRTOS

5.3 Supported Chipmakers

wolfSSL has support for chipsets from chipmakers, including: ARM, Intel, ST (STM32F2/F4), Motorola, mbed, Freescale, Microchip (PIC32), Texas Instruments, and more.

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 occured. 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 6 functions:
1. ECC sign callback
2. ECC verify callback
3. RSA sign callback
4. RSA verify callback
5. RSA encrypt callback
6. RSA decrypt callback

These two functions are prototyped by **CallbackEccSign**, **CallbackEccVerify**, **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 (*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_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_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 myEccSignCb(), myEccVerifyCb(), myRsaSignCb(), myRsaVerifyCb(), myRsaEncCb(), and myRsaDecCb(). 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.

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.

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.

```
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.

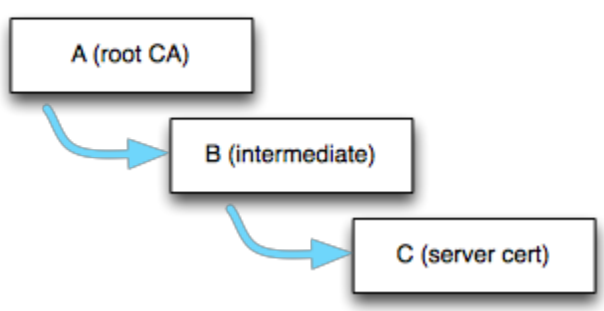
```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                                    const char *keyFile,
                                  int type);
```

keyFile is the private key file, and **type** is the format of the private key (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. 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 X509 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*, const unsigned char*,
```

```
                                  long, int)
int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX*, const unsigned
                                      char*, long, int)
int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX*, const unsigned
                                     char*, long, int)
int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX*,
                                           const unsigned char*,long)
```

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

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 either **USE_CERT_BUFFERS_1024** or **USE_CERT_BUFFERS_2048**.

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 doesn't currently have the capability to generate a standalone RSA public key. The private key can be used as both a private and public key by wolfSSL as used in test.c.

The reasoning behind the lack of individual RSA public key generation in wolfSSL is that the private key and the public key (in the form of a certificate) is all that is typically needed for SSL.

A separate public key can be loaded into wolfSSL manually using the RsaPublicKeyDecode() function if need be.

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** (randPomly 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 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_Deubgging_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 (formerly CTaoCrypt) Usage Reference

wolfCrypt is the cryptography library primarily used by wolfSSL. It is optimized for speed, small footprint, and portability. wolfSSL interchange 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 broken. 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 ***InitMd4()*** call. Use ***Md4Update()*** to update the hash and ***Md4Final()*** to retrieve the final hash

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

Md4 md4;
InitMd4(&md4);

Md4Update(&md4, buffer, sizeof(buffer));  // can be called again and again
Md4Final(&md4, md4sum);
```

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

10.1.2 MD5

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 ***InitMd5()*** call. Use ***Md5Update()*** to update the hash and ***Md5Final()*** to retrieve the final hash

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

Md5 md5;
InitMd5(&md5);

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

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

10.1.3 SHA / 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 ***InitSha()*** call. Use ***ShaUpdate()*** to update the hash and ***ShaFinal()*** to retrieve the final hash:

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

Sha sha;
InitSha(&sha);

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

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

To use either 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-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 ***InitBlake2b()*** call. Use ***Blake2bUpdate()*** to update the hash and ***Blake2bFinal()*** to retrieve the final hash:

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

Blake2b b2b;
InitBlake2b(&b2b, 64);

Blake2bUpdate(&b2b, input, sizeof(input));
Blake2bFinal(&b2b, digest, 64);
```

The second parameter to 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 ***InitRipeMd()*** call. Use ***RipeMdUpdate()*** to update the hash and ***RipeMdFinal()*** to retrieve the final hash

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

RipeMd ripemd;
InitRipeMd(&ripemd);

RipeMdUpdate(&ripemd, buffer, sizeof(buffer));  // can be called again
and again
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 ***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];

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

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

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 ***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];

GmacSetKey(&gmac, key, sizeof(key));
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 ***Poly1305SetKey()***. The process of setting a key in Poly1305 should be done again, with a new key, when next using Poly1305 after 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];

Poly1305SetKey(&pmac, key, sizeof(key));
Poly1305Update(&pmac, buffer, sizeof(buffer));
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). Supported AES modes include CBC, CTR, GCM, and CCM-8.

CBC mode is supported for both encryption and decryption and is provided through the **AesSetKey**(), **AesCbcEncrypt()** and **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
AesSetKey(&enc, key, sizeof(key), iv, AES_ENCRYPTION);
AesCbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

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 **AesCtrEncrypt()** function.

GCM mode is available for both encryption and decryption through the **AesGcmSetKey**(), **AesGcmEncrypt**(), and **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 **AesCcmSetKey**(), **AesCcmEncrypt**(), and **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 ***Des_SetKey()*** or ***Des3_SetKey()***. CBC encryption/decryption is provided through ***Des_CbcEnrypt()*** / ***Des_CbcDecrypt()*** and ***Des3_CbcEncrypt()*** / ***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
Des3_SetKey(&enc, key, iv, DES_ENCRYPTION);
Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

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

plain now contains the original plaintext from the cipher text.

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 ***CamelliaSetKey()***. CBC encryption/decryption is provided through ***CamelliaCbcEnrypt()*** and ***CamelliaCbcDecrypt()*** while direct encryption/decryption is provided through **CamelliaEncryptDirect()** and **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
Arc4SetKey(&enc, key, sizeof(key));
Arc4Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

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

plain now contains the original plaintext from the cipher text.

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

cipher now contains the cipher text from the plain text.

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

plain now contains the original plaintext from the cipher text.

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 bytes keys (128 bits) but uses 16 bytes vs (128 bits) unlike RABBIT.

```
HC128 enc;
HC128 dec;

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

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

```
byte cipher[37];

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

cipher now contains the cipher text from the plain text.

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

plain now contains the original plaintext from the cipher text.

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
Chacha_SetKey(&enc, key, keySz);
Chacha_SetIV(&enc, iv, counter); //counter is the start block
                               //counter is usually set as 0
Chacha_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the cipher text from the plain text.

```
// decrypt
Chacha_SetKey(&enc, key, keySz);
```

```
Chacha_SetIV(&enc, iv, counter);
Chacha_Process(&enc, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the cipher text.

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 ie 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;

InitRng(&rng);

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

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

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

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

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

Now to decrypt out:

```
RsaKey rsaPrivateKey;

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

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

byte plain[128];

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

Now plain will hold plainSz bytes or an error code. For complete examples of each type in wolfCrypt please see the file wolfcrypt/test/test.c. Note that the 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**".

```
InitSha(&sha);
ShaUpdate(&sha, message, sizeof(message));
ShaFinal(&sha, hash);
```

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

```
InitDsaKey(&key);
DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key, sizeof(dsaKeyBuffer));

InitRng(&rng);
```

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

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

To verify the signature, use **DsaVerify()**. If verification is successful, answer will be equal to "**1**". Once finished, free the DSA key structure using **FreeDsaKey()**.

```
DsaVerify(hash, signature, &key, &answer);
FreeDsaKey(&key);
```

Chapter 11: SSL Tutorial

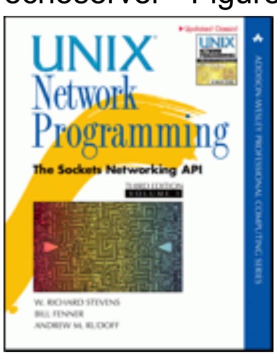
11.1 Introduction

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 1.2. wolfSSL supports SSL 3.0, TLS 1.0, 1.1, and 1.2 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:
/include
 (Common header file [Modified from unp.h in the book])
/lib
 (Common library functions)
/finished_src
     /echoclient
          (The completed echoclient code)
     /echoserver
     (The completed echoserver code)
/original_src
```

```
      /echoclient
      (The starting echoclient code)
      /echoserver
      (The starting echoserver code)
wolfSSL_SSL_Tutorial.pdf
```

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-256  test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA256 test passed!
ARC4  test passed!
DES   test passed!
DES3  test passed!
AES   test passed!
RANDOM   test passed!
RSA   test passed!
DH    test passed!
PWDBASED test passed!
OPENSSL  test passed!
peer's cert info:
issuer :
/C=US/ST=Oregon/L=Portland/O=yaSSL/OU=Programming/CN=www.yassl.com/emailAddre
ss=info@yassl.com
subject:
/C=US/ST=Oregon/L=Portland/O=yaSSL/OU=Programming/CN=www.yassl.com/emailAddre
ss=info@yassl.com
serial number:87:4a:75:be:91:66:d8:3d
SSL version is TLSv1.2
peer's cert info:
issuer :
/C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.yassl.com/emailAdd
ress=info@yassl.com
subject:
/C=US/ST=Montana/L=Bozeman/O=yaSSL/OU=Support/CN=www.yassl.com/emailAddress=i
nfo@yassl.com
SSL cipher suite is TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
```

```
serial number:02
SSL version is TLSv1.2
SSL cipher suite is TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
9a104af3d164e37c0dabe3316f7bc35b9be5fd771a09ea1cda478d4057f0b06e  input
9a104af3d164e37c0dabe3316f7bc35b9be5fd771a09ea1cda478d4057f0b06e  /tmp/output

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

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_methods();     // 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
                                   SSLv3 or 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. 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 server 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. 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));
```

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 an 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:

```
/* Define a signal handler for when the user closes the program
     with Ctrl-C. Also, turn off SA_RESTART so that the OS doesn't
     restart the call to accept()after the signal is handled. */

act.sa_handler = sig_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGINT, &act, &oact);
```

The echoserver's sig_handler function is shown below:

```c
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 the page.

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 applied 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 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 on to 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.

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 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 1.2 with DTLS. The yaSSL 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 who actively works to improve and expand wolfSSL. The yaSSL team is based in Bozeman, MT, Portland, OR, and Seattle, WA, and is always willing to help.

6. wolfSSL is the leading SSL 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 yaSSL 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 yassl!";
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 is 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

The founders and employees of wolfSSL (formerly CyaSSL) believe in Open Source Software. As such, the source code for wolfSSL is available for all to use, modify, test and enjoy under the GPL. wolfSSL 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).

We do not reserve features! As such, everything available in the commercial version of wolfSSL is also available in the open source GPL version. For more information on our licensing, please see our website or contact **info@wolfssl.com**.

14.2 Commercial Licensing

Businesses and enterprises who wish to incorporate wolfSSL into proprietary appliances or other commercial software products for re-distribution must license commercial versions. Commercial licenses for wolfSSL and yaSSL are available for $5,000 USD per product. Licenses are generally issued for one product and include unlimited distribution.

14.3 Support Packages

Support packages for wolfSSL are available on an annual basis directly from yaSSL. 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 for more details (http://www.wolfssl.com/yaSSL/Support/support_tiers.php).

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.

yaSSL Forums: http://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, 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 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 Freshmeat, follow us on Facebook, or follow our daily blog.

wolfSSL on Freshmeat	http://freshmeat.net/projects/wolfssl/
wolfSSL on Twitter	http://twitter.com/wolfSSL
wolfSSL on Facebook	http://www.facebook.com/wolfSSL
Daily Blog	http://wolfssl.com/yaSSL/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_ERROR is an error that may be 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 done 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>

void 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:
No return value for this function.

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_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.

Return Values:

0 - will be returned upon success.

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

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.

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_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_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_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

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

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) {
      // unable 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

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_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_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 flag);

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().

flag - whether or not to use the older version of setting up the information for poly1305. Passing a flag value of one 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:

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
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:

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
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

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, **SSL_FAILURE** 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).

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).

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

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)(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)(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);

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

Example:

```
See wolfssl/test.h myTicketEncCb() used by the example server and example
echoserver.
```

See Also:
wolfSSL_CTX_set_TicketHint

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_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(myLogCallback);
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 users 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 users 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 users 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 users 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 users 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 users 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

See 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 users 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 users 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()

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.

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_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.

e - 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

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 unsigned char* 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 unsigned char* 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 unsigned char* 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 unsigned char* 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 unsigned char* 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()

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 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 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 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 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 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

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

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* CAfile,
const char* CApath);

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.

Parameters:

cm - a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

CAfile - pointer to the name of the file containing CA certificates to load.

CApath - 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 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* cert,
int format);

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().

cert - 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_CertManagerVerify(WOLFSSL_CERT_MANGER* cm,
const unsigned char* buff, int 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

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.

wolfSSL_X509_get_serial_number

Synopsis:
#include <wolfssl/ssl.h>

int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509, unsigned char* 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>

unsigned char* 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* derBuffer, 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

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, unsigned char type, const void* data, unsigned short 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, unsigned short 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, unsigned char 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>

unsigned short wolfSSL_SNI_GetRequest(WOLFSSL *ssl, unsigned char 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 unsigned char* clientHello, unsigned int helloSz, unsigned char type, unsigned char* sni, unsigned int* 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, unsigned char 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, unsigned char 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, unsigned short 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, unsigned short 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

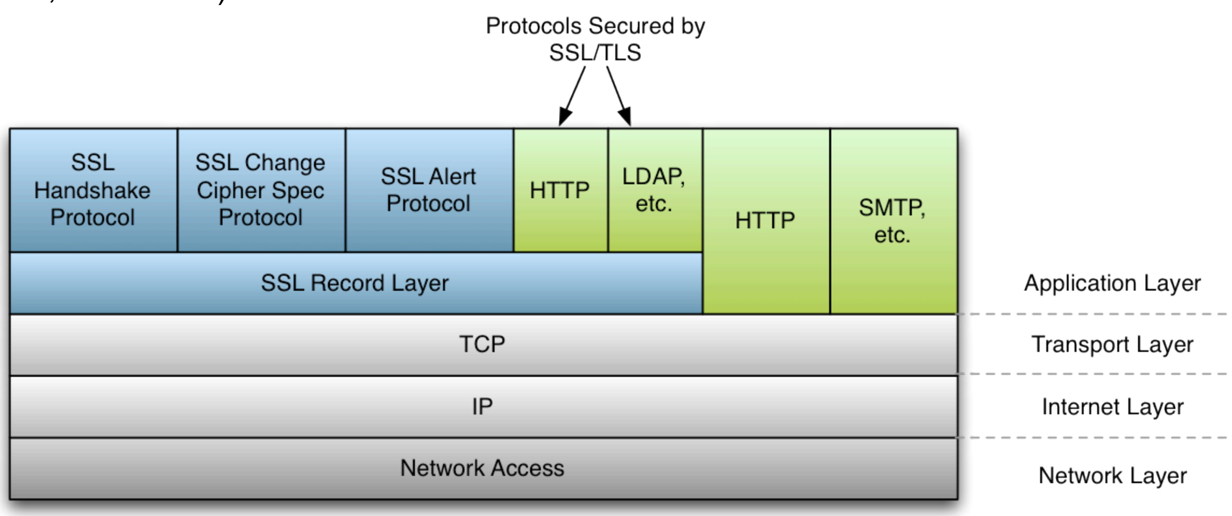
Appendix A: SSL/TLS Overview

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). 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.

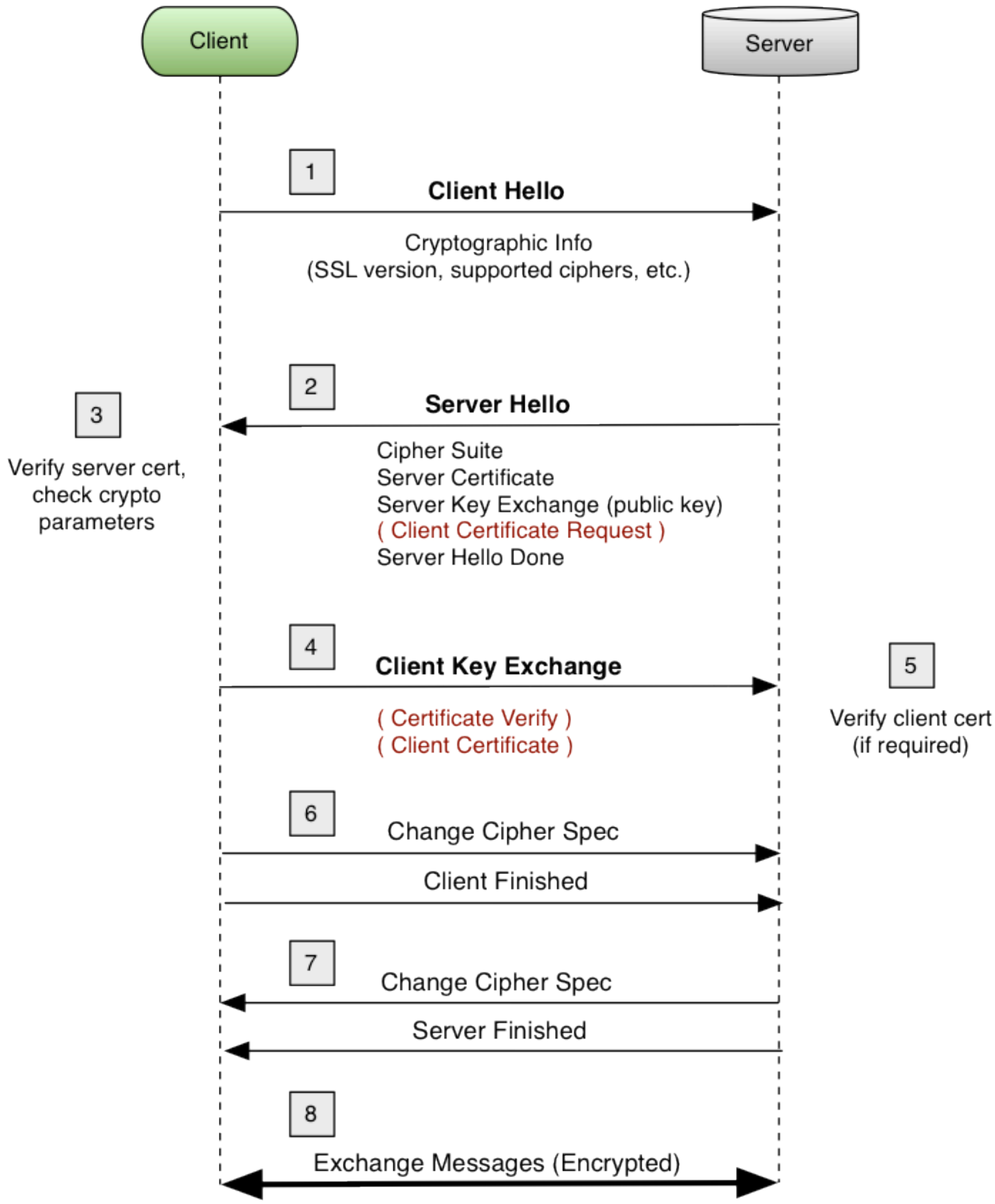
A general diagram of how the SSL protocol fits into existing protocols can be seen in **Figure 1**. SSL sits in between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the transport medium. Application protocols are layered 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.

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
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
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
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.