CyaSSL Manual

12.23.2010
Version 1.0

Table of Contents

Chapter 1: Introduction

yaSSL focuses on providing embedded security solutions with an emphasis on speed and size. With dual-licensed products to cater to a diversity of users ranging from the hobbyist to the user with commercial needs, yaSSL is happy to help customers and community members in any way we can. Our products are Open Source, giving customers the freedom to look under the hood.

CyaSSL is a C-language-based SSL library targeted for embedded and RTOS environments, primarily because of its small size and speed. CyaSSL is commonly used in applications for standard operating environments as well because of its royalty free pricing and cross platform support. CyaSSL supports the industry standards up to the current TLS 1.2 level, and is up to 20 times smaller than OpenSSL.

This manual is written as a technical guide to the CyaSSL embedded SSL library. It will explain how to obtain support, licensing details, a quick overview of SSL and TLS, details on building and integrating CyaSSL as well as a reference for CTaoCrypt and CyaSSL.

Chapter 2: Obtaining Support

I. Where to Find Help and Information

For product support, we maintain an online forum for the yaSSL related product family. Please post to our forums or contact us at **support@yassl.com** with any questions.

yaSSL Forums (http://www.yassl.com/forums)

For information regarding our products, questions regarding licensing, or general comments, please contact us at **info@yassl.com**.

II. Bugs Reports and Support Issues

If you are submitting a bug report or asking about a problem you are encountering, please include the following information with your submission:

1. CyaSSL 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. We value your feedback and make it a priority to get back to you as soon as possible.

Chapter 3: Licensing

I. Open Source

The founders and employees of yaSSL believe in Open Source Systems. As such, the source code for CyaSSL is available for all to use, modify, test and enjoy under the GPL. CyaSSL may be modified to the needs of the user as long as the user adheres to version two of the GPL License. The GPL 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 our commercial version is also available in our GPL version. For more information on our licensing, please see our web site or contact **info@yassl.com**.

II. Commercial Licensing

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

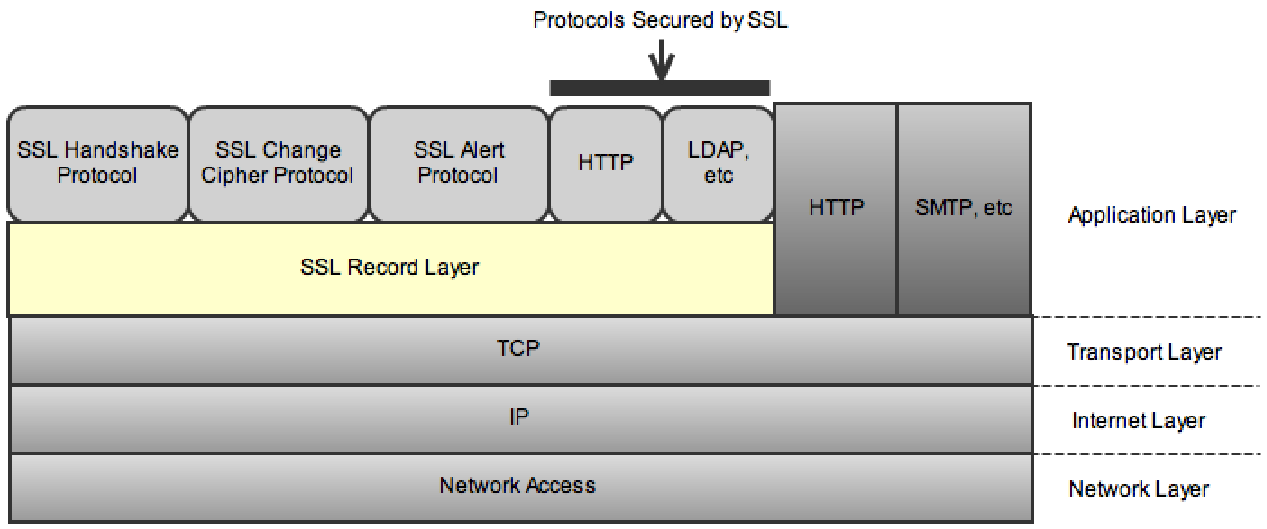
III. FLOSS Exception Details

We want specified **Free/Libre and Open Source Software** ("FLOSS") applications to be able to use specified GPL-licensed yaSSL libraries despite the fact that not all FLOSS licenses are compatible with version 2 of the GNU General Public License. Please read more about our FLOSS Exception in **Appendix A**.

Chapter 4: SSL/TLS Overview

I. General Architecture

The 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. The TLS protocol is implemented as defined in RFC 5246. 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 below in **Figure 1**.



(Figure 1: SSL Protocol Diagram)

II. Differences between SSL and TLS Protocol Versions

Secure Socket Layer (SSL) and Transport Security Layer (TLS) are both cryptographic protocols which provide secure communication over networks. These different versions are all in widespread use today in applications such as web browsing, e-mail, instant messaging and VoIP, and each is slightly different from the others.

CyaSSL supports these protocols to best suit your needs and requirements. Below you will find both an explanation and the major differences between the different SSL and TLS protocol versions.

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

Netscape's Original SSL 3.0 Draft: http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt

Comparison of SSLv2 and SSLv3: http://stason.org/TULARC/security/ssl-talk/4-11-What-is-the-difference-between-SSL-2-0-and-3-0.html

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 inter-operate. 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

RFC 2246:
http://tools.ietf.org/html/rfc2246

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.

RFC 4346:
http://tools.ietf.org/html/rfc4346#section-1.1

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.

RFC 5246:
http://tools.ietf.org/html/rfc5246

III. Reference Information on SSL

Interesting Resources
TLS - Wikipedia
(http://en.wikipedia.org/wiki/Transport_Layer_Security)

SSL versus TLS - What's the Difference?
(http://luxsci.com/blog/ssl-versus-tls-whats-the-difference.html)
Cisco - SSL: Foundation for Web Security
(http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_1-1/ssl.html)

Protocol Specifications
SSL 3.0 Specification (http://tools.ietf.org/id/draft-ietf-tls-ssl-version3-00.txt)
TLS v1.0 Specification (http://www.ietf.org/rfc/rfc2246.txt)
TLS v1.1 Specification (http://www.ietf.org/rfc/rfc4346.txt)
TLS v1.2 Specification (http://www.ietf.org/rfc/rfc5246.txt)

Technology Discussion
SSL Intro and Links
(http://www.kegel.com/ssl/)
Transport Security Layer - Wikipedia (http://en.wikipedia.org/wiki/
Transport_Layer_Security)

Chapter 5: Overview of CyaSSL

I. Overview

CyaSSL is a small and fast SSL/TLS implementation written in the C programming language. A description and general diagram of the SSL protocol can be found in **Chapter 4: SSL/TLS Overview**. CyaSSL is primarily targeted at embedded and RTOS environments because of both its speed and size, but is commonly applied in standard operating environments as well because of its royalty free pricing and cross platform portability. CyaSSL supports the industry standards up to the current TLS 1.2 level, and is up to *20 times smaller than OpenSSL*. User benchmarking and feedback reports dramatically better performance from CyaSSL over OpenSSL.

CyaSSL is built for maximum portability in ANSI standard C, and is very easy to compile on new platforms. 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 CyaSSL in another programming language we do not currently support, please contact us. The following table lists features included in the most recent release of CyaSSL.

CyaSSL Features (version 1.6.5)	**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
Minimum size of **30-100kb**, depending on build options and operating environment	Small build size for use in resource constrained environments
Runtime memory usage between **5-50kb**	Minimal dynamic memory use
DTLS support (client and server)	Streaming Multimedia
OpenSSL compatibility layer	Standard API and ease of migration from OpenSSL
MySQL integration	Wide distribution and testing

zlib compression support	Highly configurable compression support
RSA Key Generation	Fast run-time key generation support
PSK Pre-Shared Keys	Avoid RSA operations in limited environments
Simple API	Easy to learn and use
PEM and DER certificate support	No need to reconfigure certificates or keys
Intel AES-NI support	Super fast chip level encryption
Client authentication support	Use certificates to verify clients
Sniffer support	Decode SSL encrypted packets easily
Supported Algorithms - MD2, MD4, MD5, SHA-1, SHA-512, RIPEMD - DES, 3DES, AES, ARC4, RABBIT, HC-128 - RSA, DSS, DH - HMAC, PBKDF2	- Multiple hashing functions available - 3 block ciphers and 3 stream ciphers - 3 Public key options - Password based key derivation
Supported Web Servers - GoAhead, Mongoose, Lighttpd et al	Multiple lightweight embedded web server options. CyaSSL is also used in the yaSSL Embedded Web Server.

(Table 1: CyaSSL Features)

II. Supported Cipher Suites

The following cipher suites are supported by the CyaSSL embedded SSL library. 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). They **key exchange algorithm** (RSA, DSS, DH) 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-512, RIPEMD) is a hash function used to create the message digest.

CyaSSL Cipher Suites (version 1.6.0)	
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_PSK_WITH_AES_256_CBC_SHA TLS_PSK_WITH_AES_128_CBC_SHA	TLS Cipher Suites
SSL_RSA_WITH_RC4_128_SHA SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSL Cipher Suites
TLS_RSA_WITH_HC_128_CBC_MD5 TLS_RSA_WITH_HC_128_CBC_SHA TLS_RSA_WITH_RABBIT_CBC_SHA	TLS Cipher Suites with Stream Ciphers

(Table 2: CyaSSL Cipher Suites)

III. Choosing the Correct yaSSL Technology

Getting started with either CyaSSL or yaSSL begins with evaluating your project and usage needs. Common requirements for an SSL library include:

1. Small footprint!
2. Best available performance (Performance is a high priority for many, but it almost always comes with trade offs)
3. Direct access to the crypto library API
4. Specific Operating System and chipset requirements
5. OpenSSL compatibility
6. Compiler Restrictions
7. Host language preferences
8. Interest and ability to use assembly level optimizations
9. Hardware being used

To meet these needs, we offer multiple product options. These options include:

1. CyaSSL: Our full featured C based embedded SSL implementation.
2. yaSSL: Our C++ embedded SSL implementation.
3. CTaoCrypt / TaoCrypt: Our bundled or stand alone cryptography libraries.
4. Our OpenSSL compatibility layer.
5. Custom performance optimizations for specialized hardware through our consulting offerings.

CyaSSL is the optimal choice when minimal size and maximum performance in an embedded environment is the top priority. The most active development is taking place on CyaSSL with new feature additions and optimizations. yaSSL may be your best choice if you are a C++ aficionado and like to work under the hood. CTaoCrypt and TaoCrypt provide our standalone crypto library, which is callable from within CyaSSL and yaSSL. The OpenSSL compatibility layer is included with both CyaSSL and yaSSL as well.

IV. Why Developers Choose CyaSSL

There are many reasons to choose CyaSSL as your embedded SSL solution. The top reasons include size (CyaSSL can be built as small as 30k), CyaSSL supports the newest standards: TLS 1.1 and 1.2, DTLS, and Stream Ciphers, it is multi-platform, royalty free, and contains an OpenSSL compatibility API to ease porting into older applications. For a complete feature list, see the CyaSSL Product Page (http://yassl.com/yaSSL/Products_cyassl.html).

We continue to improve and add to the CyaSSL feature set. If you are looking for something we don't currently offer, please contact us.

V. 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.

CyaSSL (http://freshmeat.net/projects/cyassl/)
TaoCrypt (http://freshmeat.net/projects/taocrypt/)
yaSSL on Twitter (http://twitter.com/CyaSSL)
yaSSL on Facebook (http://www.facebook.com/pages/YaSSL/147081235315602)
Daily Blog (http://yassl.com/yaSSL/News/News.html)

Chapter 6: Building CyaSSL

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

I. Downloading CyaSSL

CyaSSL may be downloaded from the yaSSL website Download page as a ZIP file:

http://yassl.com/yaSSL/Download.html

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; Macintoshes 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.)"

II. Building on *nix

When building CyaSSL on Linux, *BSD, OS X, Solaris, or other *nix like systems, use the autconf system. To build CyaSSL you only need to run two commands:

```
./configure
make
```

To install CyaSSL run:

```
make install
```

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

```
sudo make install
```

To test the build run the testsuite program from the testsuite directory.

III. Windows

MSVC 6: Because some developers still use MSVC6, CyaSSL includes project files and a workspace for it. Though it is now deprecated and no longer supported.

VS 2005 / VS 2008: Solutions are included for Visual Studio 2005/2008 in the root directory of the install.

To test each build choose Build All and then run the testsuite program.

IV. Building in a non-standard environment

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

1. Place all of the .c files from src/ and ctaocrypt/src into the same directory.

2. Place all of the .h files from include/ and ctaocrypt/include into the same directory as above.

3. Create an openssl directory below the directory containing the files above and place all the .h files from include/openssl into the openssl directory.

4. Even though all of the CyaSSL headers are now in the same directory as the source files some build systems will still want to explicitly know where the header files are so you may need to specify that.

5. CyaSSL defaults to a little endian system unless the configure process detects big endian. Since you aren't using the configure process you'll need to define **BIG_ENDIAN_ORDER** if you are using a big endian system.

6. CyaSSL 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**.

7. Try and build the library, and let us know if you run into any problems. If you need help, then contact us at *info@yassl.com*.

8. Some defines that can modify the build are listed below:

CYASSL_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 the "CyaSSL Extensions Reference" section below for more information.

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

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.

NO_* removes parts from the build, you can also remove the respective source file as well from the build but not the header file.

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

NO_DES 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_RABBIT and **NO_HC128** remove stream cipher extensions from the build.

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

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

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

OPENSSL_EXTRA builds even more OpenSSL compatibility into the library. It is off by default.

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

NO_CYASSL_SERVER likewise removes calls specific to the server side.

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_TLS turns off TLS which isn't recommended.

NO_INLINE disables the automatic inlining of small heavily used functions. Turning this on will slow down CyaSSL and actually make it bigger since these are small functions, usually much smaller then function call setup/return.

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

DEBUG_CYASSL builds in the ability to debug to steer. It is off by default.

TEST_IPV6 turns on testing of IPV6 in the test applications. CyaSSL proper is IP neutral, but the testing applications use IPV4 by default.

CYASSL_DTLS turns on the use of DTLS or datagram TLS, this isn't widely supported or used so it is off by default.

V. ./configure Options

--enable-debug	Enable CyaSSL debugging support, disabled by default

--enable-small	Enable the smallest possible build, disabled by default
--enable-singleThreaded	Enable single threaded mode, no multi thread protections
--enable-dtls	Enable CyaSSL DTLS support, this is disabled by default
--enable-opensslExtra	Enable extra OpenSSL API compatibility, increases the size
--enable-ipv6	Enable testing of IPv6, CyaSSL proper is IP neutral
--enable-fastmath	Enable fast math for BigInts, default is disabled
--enable-fasthugemath	Enable fast math + huge code for BigInt, default is disabled
--enable-bigcache	Enable a big session cache, default is disabled
--enable-hugecace	Enable a huge session cache, default is disabled
--enable-sniffer	Enable CyaSSL sniffer support
--enable-aesni	Enable CyaSSL AES-NI support
--enable-ripemd	Enable CyaSSL RIPEMD-160 support
--enable-sha512	Enable CyaSSL SHA-512 support
--enable-sessioncerts	Enable session cert storing
--enable-keygen	Enable key generation
--enable-certgen	Enable cert generation
--disable-shared	Disable the building of a shared CyaSSL library

--disable-static	Disable the building of a static CyaSSL library
--with-libz	Optionally include libz for compression
--enable-psk	Enable Pre Shared Key support
--enable-hc128	Enable streaming cipher HC-128
--enable-ntru	Enable a build with NTRU (license required)

./configure Notes

Debug - enabling debug support allows easier debugging by compiling with debug information and defining the constant **DEBUG_CYASSL** which outputs messages to **stderr**. To turn debug logging on at runtime call *CyaSSL_Debugging_ON()*. To turn debug logging off at runtime call *CyaSSL_Debugging_OFF()*.

Small - enabling the small build option will create the smallest possible CyaSSL library. This will also remove features that may be desired like TLS, HMAC, SHA-256, error strings, or others. Only use this if the default build is too big and you don't mind losing features. A full list of features which are disabled are TLS, HMAC, AES, DES3, SHA256, Error Strings, HC128, RABBIT, PSK, DSA, and DH.

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 multi threaded and only one thread at a time will be accessing the library.

DTLS - enabling DTLS support allows users of the library to also run DTLS in addition to TLS and SSL. DTLS support is still experimental so please send us any comments/questions/suggestions.

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 then enabling this will allow better support. Our OpenSSL compatibility layer is under active development, so if there is a function missing that you need, then please contact us and we'll try to help.

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

fastmath - enabling fastmath will speed up public key operations like RSA, DH, and DSA. This 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. CyaSSL 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_SIZE** in tfm.h. 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.

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 fasthughmath 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 1055 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.

sniffer - enabling sniffer support will allow the collection of SSL traffic packets as well as the ability to decrypt those packets with the correct key file.

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.

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

certgen - enables support for self-signed x509 v3 certificate generation.

disable shared - disabling the shared library build will exclude a CyaSSL shared library from being built. By default both a shared and static library are built. During testing, integration, or on limited systems you can save time and space by disabling either library from the build process.

disable static - disabling the static library build will exclude a CyaSSL static library from being built.

libz - enabling libz will allow compression support in CyaSSL from the libz library. Think twice about including this option and using it by calling *CyaSSL_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 then it does to send it raw on all but the slowest of networks.

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

HC-128 - Though we really like the speed of this 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.

NTRU - This turns on the ability for CyaSSL to use NTRU cipher suites. An NTRU license is required to build and use these. Without the NTRU library the build will fail.

Chapter 7 : Getting Started

I. General Description

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

For instructions on the build process please see **Chapter 6**, above.

II. Testsuite

The testsuite program is designed test the ability of CyaSSL and its cryptography library CTaoCrypt to run on the system. On a successful run you should see output like:

```
MD5                    test passed!
MD4                    test passed!
SHA                    test passed!
SHA-256                test passed!
HMAC                   test passed!
ARC4                   test passed!
HC-128                 test passed!
Rabbit                 test passed!
DES                    test passed!
DES3                   test passed!
AES                    test passed!
RANDOM                 test passed!
RSA                    test passed!
DH                     test passed!
DSA                    test passed!
OPENSSL                test passed!
peer's cert info:
issuer : /C=US/ST=Oregon/L=Portland/O=yaSSL/CN=www.yassl.com/
emailAddress=info@yassl.com
subject: /C=US/ST=Oregon/L=Portland/O=yaSSL/CN=www.yassl.com/
emailAddress=info@yassl.com
peer's cert info:
issuer : /C=US/ST=Oregon/L=Portland/O=sawtooth/CN=www.sawtooth-
```

```
consulting.com/emailAddress=info@yassl.com
subject: /C=US/ST=Oregon/L=Portland/O=taoSoftDev/
CN=www.taosoftdev.com/emailAddress=info@yassl.com
Client message: hello cyassl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
b88596cd2362310b2506f9d73693cefd  input
b88596cd2362310b2506f9d73693cefd  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 and rebuilding, retesting.

III. Client Example

You can use the client example found in examples/client to test CyaSSL against any SSL server. To test against secure gmail try:

```
./client gmail.google.com 443
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
SSL connect ok, sending GET...
Server response: HTTP/1.0 302 Found
Cache-Control: private
Location: http://www.google.com
Content-Type: text/html; charset=UTF-8
Content-Length: 218
Date: Tue, 16 Feb 2010 22:25:02 GMT
Server: GFE/2.0
X-XSS-Protection: 0
```

This tells the client to connect to gmail.google.com on the https port of 443 and sends a generic GET. 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 CyaSSL default port of 11111. It also loads the client certificate in case the server wants to perform client authentication.

If one command line argument is given the client attempts to connect to the localhost at port 11111 the argument number of times and gives the average time in milliseconds that it took to perform SSL_connect(). For example,

```
./client 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 test.h located in /examples. The variables **yasslIP** and **yasslPort** control these settings. Rebuild all of the examples including testsuite when changing these settings otherwise the test programs won't be able to connect to each other.

IV. Server Example

The server example demonstrates a simple SSL server that performs client authentication and fails if the client doesn't present a certificate. 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 SSL_connect() will fail. The server will report an error "**-245, peer didn't send cert**".

V. 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 CyaSSL". 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:

```
./echoserver outupt.txt
```

VI. 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", "cyassl", and "quit" results in:

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

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

```
./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:

```
./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.

VII. Benchmark

The benchmark utility located in ctaocrypt/benchmark can be used to benchmark the cryptographic functionality of CTaoCrypt. Typical output may look like:

```
./benchmark
AES         5 megs took 0.043 seconds, 116.50 MB/s
ARC4        5 megs took 0.026 seconds, 194.72 MB/s
HC128       5 megs took 0.006 seconds, 901.07 MB/s
RABBIT      5 megs took 0.017 seconds, 299.11 MB/s
3DES        5 megs took 0.284 seconds,  17.62 MB/s

MD5         5 megs took 0.015 seconds, 334.59 MB/s
SHA         5 megs took 0.031 seconds, 163.16 MB/s
SHA-256     5 megs took 0.052 seconds,  96.28 MB/s

RSA 1024 encryption took 0.06 milliseconds, avg
RSA 1024 decryption took 0.61 milliseconds, avg
DH  1024 key generation  0.25 milliseconds, avg
DH  1024 key agreement   0.27 milliseconds, avg
```

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, the fastmath library, and the fasthugemath library.

VIII. Changing a Client Application to Use CyaSSL

1. Include the CyaSSL OpenSSL compatibility header

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

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

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

   becomes

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

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

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

   becomes

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

4. You can manually call SSL_connect() but that's not even necessary, the first call to SSL_read() or SSL_write() will initiate the SSL_connect() if it hasn't taken place yet.

5. Initialize CyaSSL and the SSL_CTX. You can use one SSL_CTX no matter how many SSL objects you end up creating. Basically you'll just have to load CA certificates to verify the server you're connecting to. Basic initialization looks like:

   ```
   InitCyaSSL();

   SSL_CTX* ctx;

   if ( (ctx = SSL_CTX_new(TLSv1_client_method())) == NULL) {
       fprintf(stderr, "SSL_CTX_new error.\n");
       exit(EXIT_FAILURE);
   }

   if (SSL_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 SSL object after each tcp connect and associate the file descriptor with the session:

   ```
   // after connecting to socket fd

   SSL* ssl;

   if ( (ssl = SSL_new(ctx)) == NULL) {
   ```

```
        fprintf(stderr, "SSL_new error.\n");
        exit(EXIT_FAILURE);
    }

    SSL_set_fd(ssl, fd);
```

7. Error checking. Each SSL_read() SSL_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 = SSL_get_error(ssl, 0);
ERR_error_string(err, buffer);
```

   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.

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

```
SSL_free(ssl);
```

   When you are completely done using SSL altogether you can free the SSL_CTX object by calling:

```
SSL_CTX_free(ctx);
FreeCyassl();
```

IX. Changing a Server Application to Use CyaSSL

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

```
SSL_CTX_new(TLSv1_client_method())
```

   becomes

```
SSL_CTX_new(TLSv1_server_method())
```

or even

```
SSL_CTX_new(SSLv23_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 (SSL_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 (SSL_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);
}
```

Chapter 8: Additional Features

I. Stream Ciphers

Ever wondered what the difference between a block cipher and a stream cipher was? 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, chucks 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 chucks of data, have block sizes requiring padding, and use a fixed, unvarying transformation.

Stream ciphers work well for large or small chucks 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.

SSL uses RC4 as the default stream cipher. It's a pretty good one, though it's getting a little older. There are some interesting advancements being made in the field and nearly two years ago CyaSSL 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 CyaSSL's stream ciphers should lessen or eliminate that performance doubt.

RC4, RABBIT and HC-128 stream ciphers are built by default into CyaSSL. Please see the examples or the documentation for usage. Links to these ciphers can be found below:

Stream Cipher	http://en.wikipedia.org/wiki/Stream_cipher
Block Cipher	http://en.wikipedia.org/wiki/Block_cipher
HC-128	http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf
Rabbit	http://www.cryptico.com/Files/filer/rabbit_fse.pdf
RC4	http://en.wikipedia.org/wiki/Rc4

II. AES-NI Support

AES is a key encryption standard used by governments worldwide, which CyaSSL has always supported. Intel has released a new set of instructions that is a faster way to implement AES. CyaSSL is currently 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.

We have added the functionality to CyaSSL 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 CyaSSL on a chipset that supports AES-NI, you can run your AES crypto 5-10 times faster!

References and further reading, ordered from general to specific are listed below. See the CyaSSL README for instructions on building CyaSSL with AES-NI.

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/

III. Digitally Signing and Authenticating with CyaSSL

CyaSSL 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, and even automobile-based computing systems. Because CyaSSL 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:

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

More background on code signing:

A great article on the topic at embedded.com: http://embedded.com/design/216500493?printable=true

General information on code signing:
http://en.wikipedia.org/wiki/Code_signing

IV. IPv6 Support

If you are an adopter of IPv6 and want to use an embedded SSL implementation then you may have been wondering if CyaSSL supports IPv6. The answer is yes, we do support CyaSSL running on top of IPv6. Note that our current test applications default to IPv4, so as to apply to a broader number of systems. Please see http://

www.yassl.com/yaSSL/Docs_Building_CyaSSL.html --enable-ipv6 to change the test applications to IPv6.

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

V. SSL Sniffer Support

Beginning with the CyaSSL 1.5.0 release, we have provided a build option allowing the CyaSSL embedded SSL library to be built with SSL Sniffer functionality. This means that you can collect SSL traffic packets and with the correct key file, are able to decrypt them as well. This could be useful for several reasons, including:

- 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 CyaSSL 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 CyaSSL's sniffer support and see a complete example, please see the "**snifftest**" app in the "ssSniffer/sslSnifferTest" folder from the CyaSSL 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 CyaSSL example echoserver and echoclient, the snifftest application must be started before the handshake begins between the server and client.

VI. Thread Safety

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

A client may share an SSL 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.

CyaSSL 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 CyaSSL leaves the responsibility of synchronizing shared objects in the hands of the user.

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

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

Chapter 9: Extensions Reference

I. Startup and Exit

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

II. Compression

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

III. CyaSSL Debugging

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

IV. Domain Name check for server certificate

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

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

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

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

V. No File System and using Certificates

Normally a file system is used to load private keys, certificates, and CAs. Since CyaSSL 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 CyaSSL_CTX_load_verify_buffer(SSL_CTX*, const unsigned char*,
                                  long)
int CyaSSL_CTX_use_certificate_buffer(SSL_CTX*, const unsigned
                                      char*, long, int)
int CyaSSL_CTX_use_PrivateKey_buffer(SSL_CTX*, const unsigned
                                     char*, long, int)
int CyaSSL_CTX_use_certificate_chain_buffer(SSL_CTX*,
                                           const unsigned char*,long)
```

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

VI. HandShake Callback

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

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

HandShakeCallBack is defined as:

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

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

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

VII. Timeout Callback

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

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

Where *TimeoutInfo* looks like:

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

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

PacketInfo is defined like this:

```
typedef struct packetInfo_st {
       char        packetName[MAX_PACKETNAME_SZ + 1]; /* SSL name */
       Timeval     timestamp;                /* when it occured    */
       unsigned char value[MAX_VALUE_SZ];    /* if fits, it's here */
```

```
        unsigned char* bufferValue;         /* otherwise here (non 0) */
        int          valueSz;               /* sz of value or buffer */
} PacketInfo;
```

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

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

See the client example for usage.

VIII. Pre Shared Keys

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

TLS_PSK_WITH_AES_256_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA

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

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

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

IX. TLS 1.1 and 1.2

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

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

for TLS 1.1 or for TLS 1.2:

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

X. RSA Key Generation

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

--enable-keygen

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

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

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

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

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

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

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

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

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

```
byte pem[4096];
int  pemSz = DerToPem(der, derSz, pem, sizeof(pem),
                      PRIVATEKEY_TYPE);
if (pemSz < 0)
    /* pemSz contains error */;
```

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

XI. Certificate Generation

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

--enable-certgen

or by defining CYASSL_CERT_GEN in Windows or non-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 asn.h named Cert:

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

Where CertName looks like:

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

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

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

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

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

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

Then, 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 pemCert[4096];

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

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

If you wish to create a CA signed certificate then a couple steps are required. After filling in the subject information you'll need to set the issuer information from the CA ceritifcate. 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,
                    &rng);
if (certSz < 0);
   /* certSz contains the error */;

certSz = SignCert(&myCert, derCert, sizeof(derCert), &caKey, &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.

XII. Standard Library Abstraction Layer

CyaSSL can now be built without the C standard library. Though the user will have to map the functions they wish to use instead of the C standard ones.

A. Memory Use

Most C programs use *malloc()* and *free()* for dynamic memory allocation. CyaSSL 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 CyaSSL memory functions in **types.h**.

B. string.h

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

C. math.h

CyaSSL 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 **dh.c**.

D. File System Use

By default, CyaSSL 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.

XIII. Input / Output Buffers

CyaSSL now uses small static buffers for input and output. They default to 128 bytes and are controlled by the RECORD_SIZE define in **cyassl_int.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 CyaSSL operated, with 16Kb static buffers that will never need dynamic memory, you can still get that option by defining LARGE_STATIC_BUFFERS.

If small static buffers are used and the user requests an **SSL_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 the current buffer size (and avoid dynamic memory use) can do this by defining STATIC_CHUNKS_ONLY.

XIV. CyaSSL NTRU Cipher Suites

CyaSSL has support for 4 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 CyaSSL is enabled with NTRU and the NTRU package is available then these cipher suites are built into the CyaSSL library. A CyaSSL client will have these cipher suites available without any interaction needed by the user. On the other hand, a CyaSSL 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 note whether or not to load NTRU keys and certificates. The CyaSSL package comes with test keys and certificates in the certs/ directory. ntru-cert.pem is the certificate and ntru-key.raw is the private key blob.

The CyaSSL 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 CyaSSL 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.

Chapter 10: CTaoCrypt Usage Reference

CTaoCrypt is the cryptography library primarily used by CyaSSL. It is optimized for speed, small footprint, and portability. CyaSSL can also interchange with other cryptography libraries as required.

Types used in the examples:

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

I. Hash Functions

MD5

To use MD5 include the MD5 header "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.

SHA

To use SHA include the SHA header "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.

Other Hashes

Likewise, the same procedures can be used with MD4 "m4.h" (which is outdated and considered broken) and SHA-256 "sha256.h".

II. Message Digests

CTaoCrypt currently provides HMAC for message digest needs. The structure Hmac is found in the header "hmac.h". HMAC initialization is done with ***HmacSetKey()***. 3 different types are supported with HMAC; MD5, SHA, and SHA-256. Here's an example with SHA-256.

```
Hmac    hmac;
byte  key[24];         // fill key with keying material
byte  buferr[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.

III. Block Ciphers

DES and 3DES

CTaoCrypt provides support for DES and 3DES (Des3 since 3 is an invalid leading C identifier). To use these include the header "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()*** / ***Des_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.

AES

CTaoCrypt also provides support for AES. Key sizes are 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits). CBC mode is supported for encrypt/decrypt. Please include the header "aes.h" to use AES. AES has a block size of 16 bytes and the IV should also be 16 bytes. The functions are exactly the same as DES and usage usually goes:

```
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.

IV. Stream Ciphers

ARC4

The most common stream cipher used on the internet is ARC4 and CTaoCrypt supports it through the header "arc.h". Usage is simpler than block ciphers because there is no block size and the key length can be any length. Use it like this:

```
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.

RABBIT

A newer stream cipher gaining popularity is RABBIT and you can use it with CTaoCrypt by including the header "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.

HC-128

Another new stream cipher in current use is HC-128 which is even faster than RABBIT. To use it with CTaoCrypt please include the header "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.

V. Public Key Cryptography

RSA

CTaoCrypt provides support for RSA through the header "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 an 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 ***CTaoCryptErrorString()*** to get a string describing the error that occurred.

```
void CTaoCryptErrorString(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 CTaoCrypt please see the file ctaocrypt/test.c.

Chapter 11: SSL Tutorial

I. Introduction

The CyaSSL embedded SSL library can easily be integrated with your existing application to provide enhanced communication security. Because CyaSSL is targeted at embedded and RTOS environments it offers both a small footprint and fast speeds. Minimum build sizes for CyaSSL range from 30-100kB depending on build options and operating environments.

Although CyaSSL is an embedded SSL library, it's full feature set makes it very functional in a desktop environment as well. CyaSSL was built for maximum portability and is generally very easy to compile on new platforms. For a full list of features and supported operating environments, see the product page: http://yassl.com/yaSSL/Products_cyassl.html.

This tutorial will walk you through integrating SSL into a simple application. The CyaSSL embedded SSL library will be used, along with a simple echoserver and echoclient. 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. If you would like to reference the exact base examples used from this book, they can be found in the figures listed below:

II. Source Code

The source code used in this tutorial can be downloaded from the following location. An overview of the contents of this ZIP file is shown in the **Initial Setup** section, below.

http://www.yassl.com/documentation/SSL_Tutorial.zip

III. Base Example Modifications

Because this tutorial is focused on the integration of SSL/TLS, the base examples are kept as simple as possible. Several modifications were made to the book examples in order to either increase simplicity or increase the range of platforms supported by the examples.

Modifications to the Echo Server (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.

Modifications to the Echo Client (tcpcli01.c)

- Moved ***str_cli()*** function from **str_cli.c** file into **tcpcli01.c** file.

Modifications to unp.h header

- This header was simplified to contain only what is needed for this example.

Please note that in these examples, certain functions will be capitalized. For example ***Fputs()*** and ***Writen()***. The authors of this book 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 *Unix Network Programming*.

IV. Initial Setup

Before we begin, you will need to download the starting example code (echoserver and echoclient) and install the CyaSSL embedded SSL library. As stated before, the example code used in this tutorial can be downloaded here: http://www.yassl.com/documentation/SSL_Tutorial.zip. The downloaded ZIP file has the following structure:

```
CyaSSL_SSL_Tutorial.pdf
/finished_src
    /echoclient
        (The completed echoclient code)
    /echoserver
        (The completed echoserver code)
    /include
        (Common header file [Modified from unp.h in the book])
    /lib
        (Common library functions)
/original_src
    /echoclient
        (The starting echoclient code)
    /echoserver
        (The starting echoserver code)
    /include
        (Common header file [Modified from unp.h in the book])
    /lib
        (Common library functions)
```

Next, you will need to download and install CyaSSL. Download the source code here: http://yassl.com/yaSSL/Download.html. For a full list of build options available, see the "*Building CyaSSL*" guide: http://yassl.com/yaSSL/Docs_Building_CyaSSL.html. CyaSSL was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building CyaSSL, do not hesitate to seek support through our Forums (http://www.yassl.com/forums).

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

```
./configure
make
```

To install CyaSSL, run:

```
sudo make install
```

To test the build, run the testsuite program from the "testsuite" directory in the CyaSSL download. The above installation will install CyaSSL into the **/usr/local/cyassl** directory.

Now that you have downloaded the example code and installed CyaSSL, we can begin modifying the example code to add SSL functionality. We will first begin with modifying the echo client and move on to the echo server.

V. Initial Compilation

To compile and run the example client and server code from **ssl_tutorial.zip**, you may use the included Makefiles. Change directory to either the echoclient or echoserver directory and run:

```
    make
```

The gcc command which is being used in the Makefile is shown below. In order to build the echo client without using the supplied Makefile, change directory to the "echoclient" directory and replace **tcpserv04.c** in the following line with **tcpcli01.c**:

```
    gcc ../lib/*.c tcpserv04.c -I ../include
```

This will build the current example, creating either a "echoserver" or "echoclient" application. To run the echo server, change directory to the "echoserver" directory and run:

```
    ./echoserver
```

To run the echo client you must pass in the IP address of the server, which in this case will be 127.0.0.1. Change directory to the "echoclient" directory and run:

```
    ./echoclient 127.0.0.1
```

VI. Libraries

We will be using the CyaSSL library for our SSL functionality. The compiled CyaSSL library is called ***libcyassl***. Unless otherwise configured, CyaSSL creates both shared and static libraries under:

```
    /usr/local/cyassl/lib
```

Modifying our GCC command (using the echo server as an example), we now have the following command. Here we include the CyaSSL include directory(**/usr/local/cyassl/include**), link to the CyaSSL library, and tell the compiler where to find the CyaSSL library using the **-L** option. Note that by using **-lcyassl** the compiler will automatically choose the correct type of library (static or shared):

```
    gcc -Wall ../lib/*.c tcpserv04.c -I ../include -I /usr/local/
cyassl/include -L/usr/local/cyassl/lib -lm -lcyassl
```

VII. Headers

Now that we're done setting up our environment and build options, we can turn our attention to the echo client example. The first thing we will need to do is include the

CyaSSL OpenSSL compatibility header. Open ***tcpcli01.c*** and add the following include to the client:

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

VIII. Startup/Shutdown

Before we can use CyaSSL, we need to initialize CyaSSL and the SSL_CTX. CyaSSL is initialized by calling the ***InitCyaSSL()*** function.

The **SSL_CTX** (SSL Context), structure contains global values for multiple SSL connections and certificate information. You can use one SSL_CTX no matter how many SSL objects you end up creating. We need to load the CA certificate into the SSL_CTX so that we can verify the server we will be connecting to.

Create a new SSL_CTX using the ***SSL_CTX_new()*** function. This function requires an argument which defines which protocol we want to use. For protocols, we have several options. CyaSSL currently supports SSLv3, TLSv1, TLSv1.1, TLSv1.2, and DTLS. Each of these have a corresponding function that we could use as an argument to ***SSL_CTX_new()***:

```
    SSLv3_client_method();       // SSL 3
    TLSv1_client_method();       // TLS 1
    TLSv1_1_client_method();     // TLS 1.1
    TLSv1_2_client_method();     // TLS 1.2
    DTLSv1_client_method();      // DTLS
```

To load the CA certificates into the SSL_CTX, we use the ***SSL_CTX_load_verify_locations()*** function. This function requires three arguments: an SSL_CTX pointer, a certificate file, and a path value. The **path** argument points to a directory containing CA certificates in PEM format. When looking up certificates, the given **certificate file** will be searched first before the **path** location. In this case, we don't need to specify a certificate path - therefore, we use the value 0. The ***SSL_CTX_load_verify_locations*** function returns ***SSL_SUCCESS*** on success and ***SSL_FAILURE*** on failure:

```
    SSL_CTX_load_verify_locations(SSL_CTX* ctx, const char* file,
                                  const char* path)
```

Putting these three things together, we have the following. Here, we chose to use TLSv1:

```
InitCyaSSL();     // Initialize CyaSSL
SSL_CTX* ctx;

/* Create the SSL_CTX */
if ( (ctx = SSL_CTX_new(TLSv1_client_method())) == NULL){
     fprintf(stderr, "SSL_CTX_new error.\n");
     exit(EXIT_FAILURE);
}

/* Load CA certificates into SSL_CTX */
if (SSL_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);
}
```

The above code should be added to the beginning of **tcpcli01.c** after both the variable definitions and the check that the user has started the client with an IP address. A version of the completed code is also included in the **ssl_tutorial.zip** file.

Now that we have initialized CyaSSL and the SSL_CTX, we also need to make sure we free the SSL_CTX object and CyaSSL when we are completely done using SSL altogether. These two lines should be placed at the end of our echo client ***main()*** function right before we call ***exit(0)***:

```
SSL_CTX_free(ctx);
FreeCyaSSL();
```

IX. SSL Object

We need to create an SSL object after each TCP Connect and associate the socket file descriptor with the session. In our echo client example, we want to do this after our call to ***Connect()***, shown below:

```
/* Connect to socket file descriptor */
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

A new SSL object is created using the ***SSL_new()*** function. This function returns a pointer to the SSL object if successful or NULL in the case of failure:

```
/* Create SSL object */
SSL* ssl;
```

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

    SSL_set_fd(ssl, sockfd);
```

X. Sending Data

We now have our foundations set up and initialized and need to start sending data securely. The echo client example uses the functions ***Writen()*** and ***Readline()*** to send and receive data between it and the echo server. We replace these calls with calls to CyaSSL's ***SSL_write()*** and ***SSL_read()*** functions.

Take note that in the echo client example, the ***main()*** function hands off the sending and receiving work to the ***str_cli()*** function. This is where replacements will be made. First, we will need access to our SSL object in the ***str_cli()*** function, so we need to add another argument and pass in our **ssl** variable. Because we will be using our **SSL** object, we can remove the **sockfd** parameter in the ***str_cli()*** function. The new ***str_cli()*** function signature:

```
    void
    str_cli(FILE *fp, SSL* ssl)
```

In our ***main()*** function, pass the new argument to ***str_cli()*** when it is called:

```
    str_cli(stdin, ssl);
```

Inside of the ***str_cli()*** function, we need to replace ***Writen()*** and ***Readline()*** and use our SSL object 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 ***SSL_write*** and ***SSL_read*** were successful. The authors of the Unix Programming book had written error checking into their ***Writen()*** function, which we must make up for when replacing it. We add a new **int** variable, "n" to monitor the return value of ***SSL_read*** and before printing out the contents of our buffer, ***recvline***, we mark the end of our read data with '\0':

```
    void
    str_cli(FILE *fp, SSL* ssl)
    {
        char    sendline[MAXLINE], recvline[MAXLINE];
        int   n = 0;

        while (Fgets(sendline, MAXLINE, fp) != NULL) {

            if(SSL_write(ssl, sendline, strlen(sendline)) !=
                         strlen(sendline)){
                err_sys("SSL_write failed");
            }

            if ((n = SSL_read(ssl, recvline, MAXLINE)) <= 0)
                err_quit("SSL_read error");

            recvline[n] = '\0';
            Fputs(recvline, stdout);
        }
    }
```

The last thing we need to do is free our SSL object when we are completely done with it. In the ***main()*** function, right before the line to free the SSL_CTX, we need to make a call to ***SSL_free()***:

```
    str_cli(stdin, ssl);

    SSL_free(ssl);          // Free SSL object
    SSL_CTX_free(ctx);      // Free SSL_CTX object
    FreeCyaSSL();           // Free CyaSSL
```

XI. Signal Handling

There is always the possibility that a user will close the echo client by hitting "Ctrl+C". In order for CyaSSL resources to be released, we need to catch that signal and handle the program exit ourselves. There are two things which we need to do:

- Add a signal handler function (here, we added it before the ***str_cli()*** function):

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    FreeCyaSSL();             /* Free CyaSSL */
    exit(EXIT_SUCCESS);
}
```

- Register this function as a signal handler using the ***signal()*** function. We added this directly after variable declarations in the ***main()*** method:

```
/* define a signal handler for when the user closes the
   program with Ctrl-C */

signal(SIGINT, sig_handler);
```

That's it - we have enabled our echo client with TLSv1!! We included the CyaSSL headers, initialized CyaSSL, created an SSL_CTX structure in which we chose what protocol we wanted to use, created an SSL object to use for sending and receiving data, replaced calls to ***Writen()*** and ***Readline()*** with ***SSL_write()*** and ***SSL_read()***, freed up SSL, SSL_CTX, and CyaSSL, and then made sure we handled the Ctrl+C signal.

The next section will deal with enabling TLSv1 in the echo server example.

XII. Echo Server

Enabling SSL/TLS in the echo server example is very similar to the steps above for the echo client. Follow the steps above, except when the protocol version is chosen during the creation of the SSL_CTX structure, you must use the server method instead. There are several options which may be chosen for the protocol:

```
SSLv3_server_methods();// SSLv3
TLSv1_server_method();      // TLSv1
TLSv1_1_server_method();     // TLSv1.1
TLSv1_2_server_method();     // TLSv1.2
SSLv23_server_method();// Allow clients to connect with
                                SSLv3 or TLSv1+
DTLSv1_server_method();// DTLS
```

The result should be similar to this:

```
/* Create and initialize SSL_CTX structure */
if ( (ctx = SSL_CTX_new(TLSv1_server_method())) == NULL){
    fprintf(stderr, "SSL_CTX_new error.\n");
    exit(EXIT_FAILURE);
}
```

When loading certificates into the SSL_CTX, you must also load the server certificate and key file in addition to the CA certificate:

```
if (SSL_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 (SSL_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);
}
```

The echo server makes a call to the ***str_echo()*** whereas the client made a call to ***str_cli()*** to handle reading and writing. As for the client, we need to modify ***str_echo()*** by replacing the **sockfd** parameter with a **SSL*** parameter to the function signature:

```
    void str_echo(SSL* ssl)
```

The calls to ***read()*** and ***Writen()*** need to be replaced with calls to the ***SSL_read()*** and ***SSL_write()*** functions. The modified ***str_echo()*** function including error checking of return values. Note that we have changed the type of the variable "n" from ***ssize_t*** to ***int*** to accommodate for the change from ***read()*** to ***SSL_read()***:

```
    void
    str_echo(SSL* ssl)
    {
        int                n;
        char           buf[MAXLINE];

    again:
        while ( (n = SSL_read(ssl, buf, MAXLINE)) > 0) {
            if(SSL_write(ssl, buf, n) != n) {
                err_sys("SSL_write failed");
            }
        }

        if( n < 0 )
            printf("SSL_read error = %d\n", SSL_get_error(ssl,n));

        else if( n == 0 )
            printf("The peer has closed the connection.\n");
    }
```

Like the echo client, we will need to add a signal handler for when the user closes the echo server 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 echo server loop by changing it from a ***for*** loop to a ***while*** loop:

```
while(cleanup != 1)
{
    // echo server code here
}
```

For the echo server 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 echo server would clean up resources and exit.

To define our signal handler and turn of SA_RESTART, first define the ***act*** and ***oact*** structures in the echo server ***main*** function:

```
struct sigaction     act, oact;
```

Insert the following code after variable declarations, before the call to ***InitCyaSSL()*** 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 echo server's ***sig_handler*** function is shown below:

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
```

```
        cleanup = 1;
        return;
    }
```

Once again, the completed source code can be found in the SSL Tutorial ZIP file.

XIII. Certificates

For testing purposes, feel free to use the certificates provided by CyaSSL. These can be found in the CyaSSL 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 one of the certificate authorities.

XIV. Conclusion

This SSL programming tutorial walked you through the process of integrating the CyaSSL embedded SSL library into a simple echo server and echo client application. Although this example is fairly simple, the same principles may be applied to enabling SSL your own application. The CyaSSL 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 CyaSSL source code directly from our website. Feel free to post to our support forums (www.yassl.com/forums) with any questions or comments you might have. If you would like more information about our products, please contact info@yassl.com.

Chapter 12: Best Practices for Embedded Devices

I. 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.

CyaSSL can be used in all of these steps to help ensure an embedded device has a secure unique private key. That will go a long ways towards securing the SSL connection itself.

Chapter 13: OpenSSL Compatibility

I. Compatibility with OpenSSL

The CyaSSL and yaSSL embedded SSL libraries provide an OpenSSL compatibility header, **ssl.h**, to ease the transition into using yaSSL or CyaSSL. Our test beds for OpenSSL compatibility are stunnel and Lighttpd, which means that we build both of them with CyaSSL as a way to test our OpenSSL compatibility API. Experimental versions of both packages built with CyaSSL are available on our download page.

II. Differences Between CyaSSL and OpenSSL

There are several differences between CyaSSL and OpenSSL. Listed here are the most prominent:

1. CyaSSL builds are 20-40 times smaller than OpenSSL. Hence it is much more useful in embedded SSL implementations.

2. Standards support: CyaSSL supports TLS 1.1 and 1.2. OpenSSL does not support TLS 1.1 or 1.2.

3. CyaSSL was built with securing streaming media in mind. OpenSSL was built before streaming media was popular on the Internet. As such, CyaSSL supports the latest streaming ciphers like Rabbit and HC-128 where OpenSSL does not.

4. License: CyaSSL is dual licensed under the GPLv2 and commercial license, with a company behind the commercial license. OpenSSL does not have a clear license.

5. We have tried to apply Occam’s razor as the guiding philosophy to our implementation of SSL. As such, our API focuses on the most critical and necessary functionality in order to simplify the problem. CyaSSL has 20 or so function calls and an additional 230 for our OpenSSL compatibility layer. OpenSSL has over 3,500.

6. Really old code versus relatively new code: CyaSSL was written starting in 2004. OpenSSL started in 1995. Coding standards and requirements are a lot different today. OpenSSL has a longer legacy to support and maintain.

7. The OpenSSL legacy code comes from supporting usage profiles and operating systems that are no longer mainstream. The legacy code makes OpenSSL easier to break and harder to fix.

8. OpenSSL was written as the SSL/TLS standards were being defined. OpenSSL's code went down a number of blind alleys. We had the luxury of writing our code once the standards were well settled.

III. Supported OpenSSL Structs

SSL_METHOD holds SSL version information and is either a client or server method.
SSL_CTX holds context information including certificates.
SSL holds session information for a secure connection.

IV. Supported OpenSSL Functions

The three structures 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()***.

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 CyaSSL 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 CyaSSL 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()**, CyaSSL 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.

V. x509 Certificates

Both the server and client can provide CyaSSL 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: Consulting

We offer both on and off site consulting - providing feature additions, porting, a Competitive Upgrade Program, and design consulting.

I. 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.

II. Competitive Upgrade Program

We will help you move from an outdated or expensive SSL library to CyaSSL with low cost and minimal disturbance to your code base.

Program Outline:

1. You need to currently be using a commercial competitor to CyaSSL.
2. You will receive up to one week of on-site consulting to switch out your old SSL library for CyaSSL. 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 CyaSSL 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 CyaSSL with ease. If you are interested in learning more, then please contact us at info@yassl.com.

III. 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 CyaSSL.

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 CyaSSL.

2. Design: Looking at your system requirements and parameters we'll work closely with you to make recommendations on how to implement CyaSSL 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@yassl.com for more information.

Appendix A: FLOSS Exception

The Sawtooth Consulting Ltd. Exception for Free/Libre and Open Source Software-only Applications Using yaSSL Libraries (the "FLOSS Exception").

Version 0.2, August 31, 2006

I. Exception Intent

We want specified “Free/Libre and Open Source Software” ("FLOSS") applications to be able to use specified GPL-licensed yaSSL libraries (the "Program") despite the fact that not all FLOSS licenses are compatible with version 2 of the GNU General Public License (the "GPL").

II. Legal Terms and Conditions

As a special exception to the terms and conditions of version 2.0 of the GPL:

You are free to distribute a Derivative Work that is formed entirely from the Program and one or more works (each, a "FLOSS Work") licensed under one or more of the licenses listed below in section 1, as long as:

1. You obey the GPL in all respects for the Program and the Derivative Work, except for identifiable sections of the Derivative Work which are not derived from the Program, and which can reasonably be considered independent and separate works in themselves,

1. All identifiable sections of the Derivative Work which are not derived from the Program, and which can reasonably be considered independent and separate works in themselves,
    a. are distributed subject to one of the FLOSS licenses listed below, and
    b. the object code or executable form of those sections are accompanied by the complete corresponding machine-readable source code for those sections on the same medium and under the same FLOSS license as the corresponding object code or executable forms of those sections, and
1. any works which are aggregated with the Program or with a Derivative Work on a volume of a storage or distribution medium in accordance with the GPL, can reasonably be considered independent and separate works in themselves which are not derivatives of either the Program, a Derivative Work or a FLOSS Work.

If the above conditions are not met, then the Program may only be copied, modified, distributed or used under the terms and conditions of the GPL or another valid licensing option from Sawtooth Consulting Ltd.

i. FLOSS License List

License name	Version(s)/Copyright Date
Academic Free License	2.0
Apache Software License	1.0/1.1/2.0
Apple Public Source License	2.0
Artistic license	From Perl 5.8.0
BSD license	"July 22 1999"
Common Development and Distribution License (CDDL)	1.0
Common Public License	1.0
GNU Library or "Lesser" General Public License (LGPL)	2.0/2.1
Jabber Open Source License	1.0
MIT License (As listed in file MIT-License.txt)	-
Mozilla Public License (MPL)	1.0/1.1
Open Software License	2.0
PHP License	3.0
Python license (CNRI Python License)	-
Python Software Foundation License	2.1.1
Sleepycat License	"1999"
University of Illinois/NCSA Open Source License	-
W3C License	"2001"
X11 License	"2001"
Zlib/libpng License	-

Zope Public License	2.0

Due to the many variants of some of the above licenses, we require that any version follow the 2003 version of the Free Software Foundation's Free Software Definition (http://www.gnu.org/philosophy/free-sw.html) or version 1.9 of the Open Source Definition by the Open Source Initiative (http://www.opensource.org/docs/definition.php).

ii. Definitions

1. Terms used, but not defined, herein shall have the meaning provided in the GPL.
2. Derivative Work means a derivative work under copyright law.

iii. Applicability

This FLOSS Exception applies to all Programs that contain a notice placed by Sawtooth Consulting Ltd. saying that the Program may be distributed under the terms of this FLOSS Exception. If you create or distribute a work which is a Derivative Work of both the Program and any other work licensed under the GPL, then this FLOSS Exception is not available for that work; thus, you must remove the FLOSS Exception notice from that work and comply with the GPL in all respects, including by retaining all GPL notices. You may choose to redistribute a copy of the Program exclusively under the terms of the GPL by removing the FLOSS Exception notice from that copy of the Program, provided that the copy has never been modified by you or any third party.