wolfSSH Documentation

2025-04-30

Contents

1 Introduction **6**
1.1 Protocol Overview . . . 6
1.2 Why Choose wolfSSH? . . . 6
1.2.1 Features . . . 6

2 Building wolfSSH **7**
2.1 Getting the Source Code . . . 7
2.2 wolfSSH Dependencies . . . 7
2.3 Building with autotools . . . 7
2.4 Building on Windows . . . 8
2.4.1 User Macros for Building on Windows . . . 8
2.5 Building in a non-standard environment . . . 9
2.6 Cross Compiling . . . 9
2.7 Install to Custom Directory . . . 9

3 Getting Started **11**
3.1 Testing . . . 11
3.1.1 wolfSSH Unit Test . . . 11
3.1.2 Testing Notes . . . 11
3.2 Examples . . . 12
3.2.1 wolfSSH Echo Server . . . 12
3.2.2 wolfSSH Client . . . 12
3.2.3 wolfSSH portfwd . . . 13
3.2.4 wolfSSH scpclient . . . 13
3.2.5 wolfSSH sftpclient . . . 13
3.2.6 wolfSSH server . . . 13
3.3 SCP . . . 14
3.4 SFTP . . . 14
3.5 Shell Support . . . 15
3.6 Post-Quantum . . . 15
3.7 Certificate Support . . . 16

4 Library Design **18**
4.1 Directory Layout . . . 18

5 wolfSSH User Authentication Callback **19**
5.1 Callback Function Prototype . . . 19
5.2 Callback Function Authentication Type Constants . . . 19
5.3 Callback Function Return Code Constants . . . 20
5.4 Callback Function Data Types . . . 20
5.4.1 Password . . . 20
5.4.2 Keyboard-Interactive . . . 21
5.4.3 Public Key . . . 21

6 Callback Function Setup API **23**
6.1 Setting the User Authentication Callback Function . . . 23
6.2 Setting the User Authentication Callback Context Data . . . 23
6.3 Getting the User Authentication Callback Context Data . . . 23
6.4 Setting the Keyboard Authentication Prompts Callback Function . . . 23
6.5 Example Echo Server User Authentication . . . 23

7 Building and Using wolfSSH SFTP **25**

7.1 Building wolfSSH SFTP . . . 25
7.2 Using wolfSSH SFTP Apps . . . 25

8 Port Forwarding 27
8.1 Building wolfSSH with Port Forwarding . . . 27
8.2 Using wolfSSH Port Forwarding Example App . . . 27

9 Notes and Limitations 28

10 Licensing 29
10.1 Open Source . . . 29
10.2 Commercial Licensing . . . 29
10.2.1 Support Packages . . . 29

11 Support and Consulting 30
11.1 How to Get Support . . . 30
11.1.1 Bugs Reports and Support Issues . . . 30
11.2 Consulting . . . 30
11.2.1 Feature Additions and Porting . . . 30
11.2.2 Competitive Upgrade Program . . . 30
11.2.3 Design Consulting . . . 31

12 wolfSSH Updates 32
12.1 Product Release Information . . . 32

13 API Reference 33
13.1 Error Codes . . . 33
13.1.1 WS_ErrorCodes (enum) . . . 33
13.1.2 WS_IOerrors (enum) . . . 33
13.2 Initialization / Shutdown . . . 34
13.2.1 wolfSSH_Init() . . . 34
13.2.2 wolfSSH_Cleanup() . . . 34
13.3 Debugging output functions . . . 35
13.3.1 wolfSSH_Debugging_ON() . . . 35
13.3.2 wolfSSH_Debugging_OFF() . . . 35
13.4 Context Functions . . . 35
13.4.1 wolfSSH_CTX_new() . . . 35
13.4.2 wolfSSH_CTX_free() . . . 36
13.4.3 wolfSSH_CTX_SetBanner() . . . 36
13.4.4 wolfSSH_CTX_UsePrivateKey_buffer() . . . 36
13.5 SSH Session Functions . . . 37
13.5.1 wolfSSH_new() . . . 37
13.5.2 wolfSSH_free() . . . 37
13.5.3 wolfSSH_set_fd() . . . 38
13.5.4 wolfSSH_get_fd() . . . 38
13.6 Data High Water Mark Functions . . . 38
13.6.1 wolfSSH_SetHighwater() . . . 38
13.6.2 wolfSSH_GetHighwater() . . . 39
13.6.3 wolfSSH_SetHighwaterCb() . . . 39
13.6.4 wolfSSH_SetHighwaterCtx() . . . 39
13.6.5 wolfSSH_GetHighwaterCtx() . . . 40
13.7 Error Checking . . . 40
13.7.1 wolfSSH_get_error() . . . 40
13.7.2 wolfSSH_get_error_name() . . . 40
13.7.3 wolfSSH_ErrorToName() . . . 41

13.8 I/O Callbacks . . . 41
13.8.1 wolfSSH_SetIORecv() . . . 41
13.8.2 wolfSSH_SetIOSend() . . . 41
13.8.3 wolfSSH_SetIOReadCtx() . . . 42
13.8.4 wolfSSH_SetIOWriteCtx() . . . 42
13.8.5 wolfSSH_GetIOReadCtx() . . . 42
13.8.6 wolfSSH_GetIOWriteCtx() . . . 43
13.9 User Authentication . . . 43
13.9.1 wolfSSH_SetUserAuth() . . . 43
13.9.2 wolfSSH_SetUserAuthCtx() . . . 43
13.9.3 wolfSSH_GetUserAuthCtx() . . . 44
13.9.4 wolfSSH_SetKeyboardAuthPrompts() . . . 44
13.9.5 wolfSSH_SetKeyboardAuthCtx() . . . 44
13.10 Set Username . . . 45
13.10.1 wolfSSH_SetUsername() . . . 45
13.11 Connection Functions . . . 45
13.11.1 wolfSSH_accept() . . . 45
13.11.2 wolfSSH_connect() . . . 46
13.11.3 wolfSSH_shutdown() . . . 46
13.11.4 wolfSSH_stream_read() . . . 46
13.11.5 wolfSSH_stream_send() . . . 47
13.11.6 wolfSSH_stream_exit() . . . 48
13.11.7 wolfSSH_TriggerKeyExchange() . . . 48
13.12 Channel Callbacks . . . 48
13.12.1 Callback Function Prototypes . . . 49
13.12.2 wolfSSH_CTX_SetChannelOpenCb . . . 49
13.12.3 wolfSSH_CTX_SetChannelOpenRespCb . . . 49
13.12.4 wolfSSH_CTX_SetChannelReqShellCb . . . 50
13.12.5 wolfSSH_CTX_SetChannelReqSubsysCb . . . 50
13.12.6 wolfSSH_CTX_SetChannelReqExecCb . . . 50
13.12.7 wolfSSH_CTX_SetChannelEofCb . . . 50
13.12.8 wolfSSH_CTX_SetChannelCloseCb . . . 51
13.12.9 wolfSSH_SetChannelOpenCtx . . . 51
13.12.10 wolfSSH_SetChannelReqCtx . . . 51
13.12.11 wolfSSH_SetChannelEofCtx . . . 52
13.12.12 wolfSSH_SetChannelCloseCtx . . . 52
13.12.13 wolfSSH_GetChannelOpenCtx . . . 52
13.12.14 wolfSSH_GetChannelReqCtx . . . 52
13.12.15 wolfSSH_GetChannelEofCtx . . . 53
13.12.16 wolfSSH_GetChannelCloseCtx . . . 53
13.12.17 wolfSSH_ChannelGetSessionType . . . 53
13.12.18 wolfSSH_ChannelGetSessionCommand . . . 54
13.13 Testing Functions . . . 54
13.13.1 wolfSSH_GetStats() . . . 54
13.13.2 wolfSSH_KDF() . . . 54
13.14 Session Functions . . . 55
13.14.1 wolfSSH_GetSessionType() . . . 55
13.14.2 wolfSSH_GetSessionCommand() . . . 55
13.15 Port Forwarding Functions . . . 56
13.15.1 wolfSSH_ChannelFwdNew() . . . 56
13.15.2 wolfSSH_ChannelFree() . . . 56
13.15.3 wolfSSH_worker() . . . 56
13.15.4 wolfSSH_ChannelGetId() . . . 57
13.15.5 wolfSSH_ChannelFind() . . . 57

13.15.6 wolfSSH_ChannelRead() . . . 57
13.15.7 wolfSSH_ChannelSend() . . . 58
13.15.8 wolfSSH_ChannelExit() . . . 58
13.15.9 wolfSSH_ChannelNext() . . . 59
13.16 Key Load Functions . . . 59
13.16.1 wolfSSH_ReadKey_buffer() . . . 59
13.16.2 wolfSSH_ReadKey_file() . . . 59
13.17 Key Exchange Algorithm Configuration . . . 60
13.17.1 wolfSSH Set Algo Lists . . . 60
13.17.2 wolfSSH Get Algo List . . . 61
13.17.3 wolfSSH_CheckAlgoName . . . 61
13.17.4 wolfSSH Query Algorithms . . . 62

14 wolfSSL SFTP API Reference 63
14.1 Connection Functions . . . 63
14.1.1 wolfSSH_SFTP_accept() . . . 63
14.1.2 wolfSSH_SFTP_connect() . . . 63
14.1.3 wolfSSH_SFTP_negotiate() . . . 64
14.2 Protocol Level Functions . . . 64
14.2.1 wolfSSH_SFTP_RealPath() . . . 64
14.2.2 wolfSSH_SFTP_Close() . . . 65
14.2.3 wolfSSH_SFTP_Open() . . . 66
14.2.4 wolfSSH_SFTP_SendReadPacket() . . . 66
14.2.5 wolfSSH_SFTP_SendWritePacket() . . . 67
14.2.6 wolfSSH_SFTP_STAT() . . . 68
14.2.7 wolfSSH_SFTP_LSTAT() . . . 69
14.2.8 wolfSSH_SFTPNAME_free() . . . 69
14.2.9 void wolfSSH_SFTPNAME_free(WS_SFTPNMAE* name); . . . 70
14.3 Reget / Reput Functions . . . 70
14.3.1 wolfSSH_SFTP_SaveOfst() . . . 70
14.3.2 wolfSSH_SFTP_GetOfst() . . . 71
14.3.3 wolfSSH_SFTP_ClearOfst() . . . 72
14.3.4 wolfSSH_SFTP_Interrupt() . . . 72
14.4 Command Functions . . . 73
14.4.1 wolfSSH_SFTP_Remove() . . . 73
14.4.2 wolfSSH_SFTP_MKDIR() . . . 74
14.4.3 wolfSSH_SFTP_RMDIR() . . . 74
14.4.4 wolfSSH_SFTP_Rename() . . . 75
14.4.5 wolfSSH_SFTP_LS() . . . 75
14.4.6 wolfSSH_SFTP_Get() . . . 76
14.4.7 wolfSSH_SFTP_Put() . . . 77
14.5 SFTP Server Functions . . . 78
14.5.1 wolfSSH_SFTP_read() . . . 78

1 Introduction

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

wolfSSH is an implementation of the SSH (Secure Shell) server written in C and uses the wolfCrypt library which is also available from wolfSSL. Furthermore, wolfSSH has been built from the ground up in order for it to have multi-platform use. This implementation is based off of the SSH v2 specification.

1.1 Protocol Overview

SSH is a layered set of protocols that provide multiplexed streams of data between two peers. Typically, it is used for securing a connection to a shell on the server. However, it is also commonly used to securely copy files between two machines or tunnel the X display protocol.

1.2 Why Choose wolfSSH?

The wolfSSH library is a lightweight SSHv2 server library written in ANSI C and targeted for embedded, RTOS, and resource-constrained environments - primarily because of its small size, speed, and feature set. It is commonly used in standard operating environments as well because of its royalty-free pricing and excellent cross platform support. wolfSSH supports the industry standard SSH v2. wolfSSH is powered by the wolfCrypt library. A version of the wolfCrypt cryptography library has been FIPS 140-3 validated (Certificate #4718) and FIPS 140-2 validated (Certificate #3389). For additional information, visit the wolfCrypt FIPS FAQ or contact fips@wolfssl.com.

1.2.1 Features

- SSH v2.0 (server and client)
- Minimum footprint size of 33kB
- Runtime memory usage between 1.4 and 2kB, not including a configurable receive buffer
- Multiple hashing functions: SHA-1, SHA-2 (SHA-256, SHA-384, SHA-512)
- Block and authenticated ciphers: AES-CBC, AES-CTR, AES-GCM
- Key exchange options: DHE and ECDHE (with curves NISTP256, NISTP384, NISTP521)
- Public key authentication options: RSA and ECDSA (with curves NISTP256, NISTP384, NISTP521)
- User authentication support (password, keyboard-interactive and public key authentication)
- Simple API
- PEM and DER X.509 certificate support
- Hardware Cryptography Support: Intel AES-NI support, Intel AVX1/2, RDRAND, RDSEED, Cavium NITROX support, STM32F2/F4 hardware crypto support, Freescale CAU / mmCAU / SEC, Microchip PIC32MZ
- Post quantum hybrid key exchange with Hybrid ECDH-P256 Kyber-Level1
- Support for SFTP, SCP, SSH-AGENT, local and remote port forwarding

2 Building wolfSSH

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

This section explains how to build wolfSSH on Linux, un*x-like (BSD, macOS) and Windows environments, and provides guidance for building in a non-standard environment. You will find a getting started guide and example in section 3.

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

2.1 Getting the Source Code

The most recent, up to date version can be downloaded from the GitHub website here: https://github.com/wolfSSL/wolfssh.

Either click the "Download ZIP" button or use the following command in your terminal:

```
$ git clone https://github.com/wolfSSL/wolfssh.git
```

2.2 wolfSSH Dependencies

Since wolfSSH is dependent on wolfCrypt, a configuration of wolfSSL is necessary. wolfSSL can be downloaded here: https://github.com/wolfSSL/wolfssl. The simplest configuration of wolfSSL required for wolfSSH is the default build that can be built from the root directory of wolfSSL with the following commands:

```
$ ./autogen.sh (only if you cloned from GitHub)
$ ./configure --enable-wolfssh
$ make check
$ sudo make install
```

To use the key generation function in wolfSSH, wolfSSL will need to be configured with keygen:

```
--enable-keygen
```

If the bulk of wolfSSL code isn't desired, wolfSSL can be configured with the crypto only option:

```
--enable-cryptonly
```

2.3 Building with autotools

When building on Linux, BSD, macOS, Solaris, or other un*x-like environments, use the autotools system. To build wolfSSH run the following commands:

```
$ ./autogen.sh (only if you cloned from GitHub)
$ ./configure
$ make
$ make install
```

You can append build options to the configure command. For a list of available configure options and their purposes run:

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

To build wolfSSH run:

```
$ make
```

To ensure that wolfSSH has been built correctly, check to see if all of the tests have passed with:

```
$ make check
```

To install wolfSSH run:

```
$ make install
```

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

```
$ sudo make install
```

If you want to build only the wolfSSH library located in wolfssh/src/ and not the additional items (examples and tests) you can run the following command from the wolfSSH root directory:

```
$ make src/libwolfssh.la
```

2.4 Building on Windows

The Visual Studio project file can be found in the directory *ide\winvs*.

The solution file, 'wolfssh.sln', facilitates building wolfSSH and its example and test programs. The solution provides both Debug and Release builds of Static and Dynamic 32- or 64-bit libraries. The file user_settings.h should be used in the wolfSSL build to configure it.

This project assumes that the wolfSSH and wolfSSL source directories are installed side-by-side and do not have the version number in their names:

```
Projects\
wolfssh\
wolfssl\
```

The file `wolfssh\ide\winvs\user_settings.h` contains the settings used to configure wolfSSL with the appropriate settings. This file must be copied from the directory `wolfssh\ide\winvs` to `wolfssl\IDE\WIN`. If you change one copy you must change both copies. The option `WOLFCRYPT_ONLY` disables the build of the wolfSSL files and only builds the wolfCrypt algorithms. To also keep wolfSSL, delete that option.

2.4.1 User Macros for Building on Windows

The solution is using user macros to indicate the location of the wolfSSL library and headers. All paths are set to the default build destinations in the wolfssl64 solution. The user macro wolfCryptDir is used as the base path for finding the libraries. It is initially set to `..\..\..\..\wolfssl`. And then, for example, the additional include directories value for the API test project is set to `$(wolfCryptDir)`.

The wolfCryptDir path must be relative to the project files, which are all one directory down

```
wolfssh/wolfssh.vcxproj
unit-test/unit-test.vcxproj
```

The other user macros are the directories where the wolfSSL libraries for the different builds may be found. So the user macro 'wolfCryptDllRelease64' is initially set to:

```
$(wolfCryptDir)\x64\DLL Release
```

This value is used in the debugging environment for the echoserver's 64-bit DLL Release build is set to:

```
PATH=$(wolfCryptDllRelease64);%PATH%
```

When you run the echoserver from the debugger, it finds the wolfSSL DLL in that directory.

2.5 Building in a non-standard environment

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

1. The source and header files need to remain in the same directory structure as they are in the wolfSSH download package.
2. Some build systems will want to explicitly know where the wolfSSH header files are located, so you may need to specify that. They are located in the /wolfssh directory. Typically, you can add the directory to your include path to resolve header problems.
3. wolfSSH defaults to a little endian system unless the configure process detects big endian. Since users building in a non-standard environment aren't using the configure process, BIG_ENDIAN_ORDER will need to be defined if using a big endian system.
4. Try to build the library and let us know if you run into any problems. If you need help, contact us at support@wolfssl.com.

2.6 Cross Compiling

Many users on embedded platforms cross compile for their environment. The easiest way to cross compile the library is to use the configure system. It will generate a Makefile which can then be used to build wolfSSH.

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

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

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

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

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

```
$ make
$ sudo make install
```

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

2.7 Install to Custom Directory

To setup a custom install directory for wolfSSL use the following:

```
$ ./configure --prefix=~/wolfSSL
$ make
$ make install
```

This will place the library in ~/wolfSSL/lib and the includes in ~/wolfssl/include. To set up a custom install directory for wolfSSH and specify the custom wolfSSL library and include directories use the following:

```
$ ./configure --prefix=~/wolfssh --libdir=~/wolfssl/lib --includedir=~/
   wolfssl/include
$ make
$ make install
```

Make sure the paths above match your actual locations.

3 Getting Started

After downloading and building wolfSSH, there are some automated test and example programs to show the uses of the library.

3.1 Testing

3.1.1 wolfSSH Unit Test

The wolfSSH unit test is used to verify the API. Both positive and negative test cases are performed. This test can be run manually and it additionally runs as part of other automated processes such as the make and make check commands.

All examples and tests must be run from the wolfSSH home directory so the test tools can find their certificates and keys.

To run the unit test manually:

```
$ ./tests/unit.test
```

or

```
$ make check (when using autoconf)
```

3.1.2 Testing Notes

After cloning the repository, be sure to make the testing private keys read- only for the user, otherwise ssh_client will tell you to do it.

```
$ chmod 0600 ./keys/gretel-key-rsa.pem ./keys/hansel-key-rsa.pem \
             ./keys/gretel-key-ecc.pem ./keys/hansel-key-ecc.pem
```

Authentication against the example echoserver can be done with a password or public key. To use a password the command line:

```
$ ssh -p 22222 USER@localhost
```

Where the *USER* and password pairs are:

```
jill:upthehill
jack:fetchapail
```

To use public key authentication use the command line:

```
$ ssh -i ./keys/USER-key-TYPE.pem -p 22222 USER@localhost
```

Where the *USER* can be gretel or hansel, and TYPE is rsa or ecc.

Keep in mind, the echoserver has several fake accounts in its wsUserAuth callback function. (jack, jill, hansel, and gretel) When the shell support is enabled, those fake accounts will not work. They don't exist in the system's passwd file. The users will authenticate, but the server will err out because they don't exist in the system. You can add your own username to the password or public key list in the echoserver. That account will be logged into a shell started by the echoserver with the privileges of the user running echoserver.

3.2 Examples

3.2.1 wolfSSH Echo Server

The echoserver is the workhorse of wolfSSH. It originally only allowed one to authenticate one of the canned account and would repeat the characters typed into it. When enabling shell support, see the later section, it can spawn a user shell. It will need an actual user name on the machine and an updated user authentication callback function to validate the credentials. The echoserver can also handle SCP and SFTP connections. From the terminal run:

```
$ ./examples/echoserver/echoserver -f
```

The option `-f` enables echo-only mode. From another terminal run:

```
$ ssh jill@localhost -p 22222
```

When prompted for a password, enter "upthehill". The server will send a canned banner to the client:

```
wolfSSH Example Echo Server
```

Characters typed into the client will be echoed to the screen by the server. If the characters are echoed twice, the client has local echo enabled. The echo server isn't being a proper terminal so the CR/LF translation will not work as expected.

The following control characters will trigger special actions in the echoserver:

- CTRL-C: Terminate the connection.
- CTRL-E: Print out some session statistics.
- CTRL-F: Trigger a new key exchange.

The echoserver tool accepts the following command line options:

```
-1              exit after a single (one) connection
-e              expect ECC public key from client
-E              use ECC private key
-f              echo input
-p <num>        port to accept on, default 22222
-N              use non-blocking sockets
-d <string>     set the home directory for SFTP connections
-j <file>       load in a public key to accept from peer
```

3.2.2 wolfSSH Client

The client establishes a connection to an SSH server. In its simplest mode, it sends the string "Hello, wolfSSH!" to the server, prints the response, and then exits. With the pseudo terminal option, the client will be a real client.

The client tool accepts the following command line options:

```
-h <host>       host to connect to, default 127.0.0.1
-p <num>        port to connect on, default 22222
-u <username>   username to authenticate as (REQUIRED)
-P <password>   password for username, prompted if omitted
-e              use sample ecc key for user
-i <filename>   filename for the user's private key
-j <filename>   filename for the user's public key
-x              exit after successful connection without doing
                read/write
-N              use non-blocking sockets
```

```
-t              use psuedo terminal
-c <command>    executes remote command and pipe stdin/stdout
-a              Attempt to use SSH-AGENT
```

3.2.3 wolfSSH portfwd

The portfwd tool establishes a connection to an SSH server and sets up a listener for local port forwarding or requests a listener for remote port forwarding. After a connection, the tool terminates.

The portfwd tool accepts the following command line options:

```
-h <host>       host to connect to, default 127.0.0.1
-p <num>        port to connect on, default 22222
-u <username>   username to authenticate as (REQUIRED)
-P <password>   password for username, prompted if omitted
-F <host>       host to forward from, default 0.0.0.0
-f <num>        host port to forward from (REQUIRED)
-T <host>       host to forward to, default to host
-t <num>        port to forward to (REQUIRED)
```

3.2.4 wolfSSH scpclient

The scpclient, wolfscp, establishes a connection to an SSH server and copies the specified files from or to the local machine.

The scpclient tool accepts the following command line options:

```
-H <host>       host to connect to, default 127.0.0.1
-p <num>        port to connect on, default 22222
-u <username>   username to authenticate as (REQUIRED)
-P <password>   password for username, prompted if omitted
-L <from>:<to> copy from local to server
-S <from>:<to> copy from server to local
```

3.2.5 wolfSSH sftpclient

The sftpclient, wolfsftp, establishes a connection to an SSH server and allows directory navigation, getting and putting files, making and removing directories, etc.

The sftpclient tool accepts the following command line options:

```
-h <host>       host to connect to, default 127.0.0.1
-p <num>        port to connect on, default 22222
-u <username>   username to authenticate as (REQUIRED)
-P <password>   password for username, prompted if omitted
-d <path>       set the default local path
-N              use non blocking sockets
-e              use ECC user authentication
-l <filename>   local filename
-r <filename>   remote filename
-g              put local filename as remote filename
-G              get remote filename as local filename
```

3.2.6 wolfSSH server

This tool is a place holder.

3.3 SCP

wolfSSH includes server-side support for scp, which includes support for both copying files 'to' the server, and copying files 'from' the server. Both single file and recursive directory copy are supported with the default send and receive callbacks.

To compile wolfSSH with scp support, use the `--enable-scp` build option or define `WOLFSSL_SCP`:

```
$ ./configure --enable-scp
$ make
```

The wolfSSL example server has been set up to accept a single scp request, and is compiled by default when compiling the wolfSSH library. To start the example server, run:

```
$ ./examples/server/server
```

Standard scp commands can be used on the client side. The following are a few examples, where `scp` represents the ssh client you are using.

To copy a single file TO the server, using the default example user "jill":

```
$ scp -P 22222 <local_file> jill@127.0.0.1:<remote_path>
```

To copy the same single file TO the server, but with timestamp and in verbose mode:

```
$ scp -v -p -P 22222 <local_file> jill@127.0.0.1:<remote_path>
```

To recursively copy a directory TO the server:

```
$ scp -P 22222 -r <local_dir> jill@127.0.0.1:<remote_dir>
```

To copy a single file FROM the server to the local client:

```
$ scp -P 22222 jill@127.0.0.1:<remote_file> <local_path>
```

To recursively copy a directory FROM the server to the local client:

```
$ scp -P 22222 -r jill@127.0.0.1:<remote_dir> <local_path>
```

3.4 SFTP

wolfSSH provides server and client side support for SFTP version 3. This allows the user to set up an encrypted connection for managing file systems.

To compile wolfSSH with SFTP support, use the `--enable-sftp` build option or define `WOLFSSH_SFTP`:

```
$ ./configure --enable-sftp
$ make
```

For full API usage and implementation details, please see the wolfSSH User Manual.

The SFTP client created is located in the directory examples/sftpclient/ and the server is ran using the same echoserver as with wolfSSH.

```
src/wolfssh$ ./examples/sftpclient/wolfsftp
```

A full list of supported commands can be seen with typeing "help" after a connection.

```
wolfSSH sftp> help

Commands :
    cd  <string>                        change directory
    chmod <mode> <path>                 change mode
    get <remote file> <local file>      pulls file(s) from server
    ls                                  list current directory
    mkdir <dir name>                    creates new directory on server
    put <local file> <remote file>      push file(s) to server
    pwd                                 list current path
    quit                                exit
    rename <old> <new>                  renames remote file
    reget <remote file> <local file>    resume pulling file
    reput <remote file> <local file>    resume pushing file
    <crtl + c>                          interrupt get/put cmd
```

An example of connecting to another system would be

```
src/wolfssh$ ./examples/sftpclient/wolfsftp -p 22 -u user -h 192.168.1.111
```

3.5 Shell Support

wolfSSH's example echoserver can now fork a shell for the user trying to log in. This currently has only been tested on Linux and macOS. The file echoserver.c must be modified to have the user's credentials in the user authentication callback, or the user authentication callback needs to be changed to verify the provided password.

To compile wolfSSH with shell support, use the –enable-shell build option or define WOLFSSH_SHELL:

```
$ ./configure --enable-shell
$ make
```

By default, the echoserver will try to start a shell. To use the echo testing behavior, give the echoserver the command line option -f.

```
$ ./examples/echoserver/echoserver -f
```

3.6 Post-Quantum

wolfSSH now supports the post-quantum algorithm Kyber. It uses the NIST submission's Level 1 parameter set implemented by liboqs via an integration with wolfSSH. It is hybridized with ECDHE over the P-256 ECC curve.

In order be able to use liboqs, you must have it built and installed on your system. We support the 0.7.0 release of liboqs. You can download it from the following link:

```
https://github.com/open-quantum-safe/liboqs/archive/refs/tags/0.7.0.tar.gz
```

Once unpacked, this would be sufficient:

```
$ cd liboqs-0.7.0
$ mkdir build
$ cd build
$ cmake -DOQS_USE_OPENSSL=0 ..
$ make all
$ sudo make install
```

In order to enable support for Kyber Level1 hybridized with ECDHE over the P-256 ECC curve in wolfSSH, use the `--with-liboqs` build option during configuration:

```
$ ./configure --with-liboqs
```

The wolfSSH client and server will automatically negotiate using Kyber Level1 hybridized with ECDHE over the P-256 ECC curve if this feature is enabled.

```
$ ./examples/echoserver/echoserver -f

$ ./examples/client/client -u jill -P upthehill
```

On the client side, you will see the following output:

```
Server said: Hello, wolfSSH!
```

If you want to see inter-operability with OpenQauntumSafe's fork of OpenSSH, you can build and execute the fork while the echoserver is running. Download the release from here:

```
https://github.com/open-quantum-safe/openssh/archive/refs/tags/OQS-OpenSSH
   -snapshot-2021-08.tar.gz
```

The following is sufficient for build and execution:

```
$ tar xmvf openssh-OQS-OpenSSH-snapshot-2021-08.tar.gz
$ cd openssh-OQS-OpenSSH-snapshot-2021-08/
$ ./configure --with-liboqs-dir=/usr/local
$ make all
$ ./ssh -o"KexAlgorithms +ecdh-nistp256-kyber-512-sha256" \
  -o"PubkeyAcceptedAlgorithms +ssh-rsa" \
  -o"HostkeyAlgorithms +ssh-rsa" \
  jill@localhost -p 22222
```

NOTE: when prompted, enter the password which is "upthehill".

You can type a line of text and when you press enter, the line will be echoed back. Use CTRL-C to terminate the connection.

3.7 Certificate Support

wolfSSH can accept X.509 certificates in place of just public keys when authenticating a user.

To compile wolfSSH with X.509 support, use the `--enable-certs` build option or define `WOLFSSH_CERTS`:

```
$ ./configure --enable-certs
$ make
```

To provide a CA root certificate to validate a user's certificate, give the echoserver the command line option `-a`.

```
$ ./examples/echoserver/echoserver -a ./keys/ca-cert-ecc.pem
```

The echoserver and client have a fake user named "john" whose certificate will be used for authentication.

An example echoserver/client connection using the example certificate john-cert.der would be:

```
$ ./examples/echoserver/echoserver -a ./keys/ca-cert-ecc.pem -K john:./
    keys/john-cert.der

$ ./examples/client/client -u john -J ./keys/john-cert.der -i ./keys/john-
    key.der
```

4 Library Design

The wolfSSH library is meant to be included directly into an application. The primary use case in mind during development is replacing serial- or telnet-based menus on embedded devices. The library is agnostic to networking using I/O callbacks, but provides callbacks for *NIX and Windows networking by default as examples. Timing is platform specific and should be provided by the application, functions will be provided to perform actions on timeouts.

4.1 Directory Layout

The wolfSSH library header files are located in the **wolfssh** directory. The only header required to be included in a source file is **wolfssh/ssh.h**. An example is shown below.

```
#include <wolfssh/ssh.h>
```

The wolfSFTP library header file is also included in the wolfssh directory. To call this header file use:

```
#include <wolfssh/wolfsftp.h>
```

All main source files are located in the **src** directory that resides in the root directory.

5 wolfSSH User Authentication Callback

wolfSSH needs to be able to authenticate users connecting to the server no matter which environment the library is embedded. Lookups may need to be done using passwords or RSA public keys stored in a text file, database, or hard coded into the application.

wolfSSH provides a callback hook that receives the username, either the password or public key provided in the user authentication message and the requested authentication type. The callback function then performs the appropriate lookups and gives a reply. Providing a callback is required.

The callback should return one of several failures or a success. The library will treat all the failures the same except for logging purposes, i.e. return the User Authorization Failure message to the client who will try again.

For password lookups, the plaintext password is given to the callback function. The username and password should be checked and if they match, a success returned. On success, the SSH handshake continues immediately. Password changing is not supported at this time.

For public key lookups, the public key blob from the client is given to the callback function. The public key is checked against the server's list of valid client public keys. If the public key provided matches the known public key for that user. The wolfSSH library performs the actual validation of the user authentication signature following the process described in RFC 4252 §7.

Commonly for public keys, the server stores either the users' public keys as generated by the ssh-keygen utility or stores a fingerprint of the public key. This value for a user is what is compared. The client will provide a signature of the session ID and the user authentication request message using its private key; the server verifies this signature using the public key.

5.1 Callback Function Prototype

The prototype for the user authentication callback function is:

```
int UserAuthCb(byte authType , const WS_UserAuthData*
authData , void* ctx );
```

This function prototype is of the type:

```
WS_CallbackUserAuth
```

The parameter authType is either:

```
WOLFSSH_USERAUTH_PASSWORD
```

or

```
WOLFSSH_USERAUTH_PUBLICKEY
```

The parameter, authData, is a pointer to the authentication data.

See section 5.4 for a description of WS_UserAuthData

The parameter **ctx** is an application defined context; wolfSSH does nothing with and knows nothing about the data in the context, it only provides the context pointer to the callback function.

5.2 Callback Function Authentication Type Constants

The following are values passed to the user authentication callback function in the authType parameter. It guides the callback function as to the type of authentication data to check. A system could use either a password or public key for different users.

```
WOLFSSH_USERAUTH_PASSWORD
WOLFSSH_USERAUTH_KEYBOARD
WOLFSSH_USERAUTH_PUBLICKEY
```

5.3 Callback Function Return Code Constants

The following are the return codes the callback function shall return to the library. The failure code indicates that nothing was done and the callback couldn't do any checking.

The invalid codes indicate why the user authentication is being rejected:

```
invalid username
invalid password
invalid public key
```

The server indicates *success* or *failure* to the client, the specific failure type is only used for logging. There is a special success and failure response the callback can return to the library, *partial-success*. This means the authentication type was successful, but another authentication type is still required to fully authenticate. The server will send a user authentication failure message with the partial-success flag set to client.

```
WOLFSSH_USERAUTH_SUCCESS
WOLFSSH_USERAUTH_FAILURE
WOLFSSH_USERAUTH_INVALID_USER
WOLFSSH_USERAUTH_INVALID_PASSWORD
WOLFSSH_USERAUTH_INVALID_PUBLICKEY
WOLFSSH_USERAUTH_PARTIAL_SUCCESS
WOLFSSH_USERAUTH_SUCCESS_ANOTHER
```

5.4 Callback Function Data Types

The client data is passed to the callback function in a structure called `WS_UserAuthData`. It contains pointers to the data in the message. Common fields are in this structure. Method specific fields are in a union of structures in the user authentication data.

```
typedef struct WS_UserAuthData {
    byte authType ;
    byte* username ;
    word32 usernameSz ;
    byte* serviceName ;
    word32 serviceNameSz ;
    union {
        WS_UserAuthData_Password password ;
        WS_UserAuthData_PublicKey publicKey ;
        WS_UserAuthData_Keyboard keyboard ;
    } sf;
} WS_UserAuthData;
```

5.4.1 Password

The `username` and `usernameSz` parameters are the username provided by the client and its size in octets.

The `password`and `passwordSz` fields are the client's password and its size in octets.

While set if provided by the client, the parameters hasNewPassword, newPassword, and newPasswordSz are not used. There is no mechanism to tell the client to change its password at this time.

```
typedef struct WS_UserAuthData_Password {
    uint8_t* password ;
    uint32_t passwordSz ;
    uint8_t hasNewPasword ;
    uint8_t* newPassword ;
    uint32_t newPasswordSz ;
} WS_UserAuthData_Password;
```

5.4.2 Keyboard-Interactive

The Keyboard-Interactive mode allows for an arbitrary number of prompts and responses from the server to the client. The structure that contains the information is as follows:

```
typedef struct WS_UserAuthData_Keyboard {
    word32 promptCount;
    word32 responseCount;
    word32 promptNameSz;
    word32 promptInstructionSz;
    word32 promptLanguageSz;
    byte* promptName;
    byte* promptInstruction;
    byte* promptLanguage;
    word32* promptLengths;
    word32* responseLengths;
    byte* promptEcho;
    byte** responses;
    byte** prompts;
} WS_UserAuthData_Keyboard;
```

On the client side, during authentication, the promptName and promptInstruction will indicate to the user information about the authentication. The promptLanguage field is a deprecated part of the API and is ignored.

The promptCount indicates how many prompts there are. prompts contains the array of prompts, promptLengths is an array containing the length of prompt in prompts. promptEcho in an array of booleans indicating whether or not each prompt response should be echoed to the user as they are typing.

Conversely there is responseCount to set the number of responses given. responses and responseLengths contain the response data for the prompts.

The server can set the prompts using the wolfSSH_SetKeyboardAuthPrompts() callback. The WS_CallbackKeyboardAuthPrompts callback should set the promptCount, prompts, promptLengths and promptEcho. The other prompt* items are optional.

The server should return WOLFSSH_USERAUTH_SUCCESS_ANOTHER from the WS_CallbackUserAuth callback to execute subsequent request / response rounds.

5.4.3 Public Key

wolfSSH will support multiple public key algorithms. The publicKeyType member points to the algorithm name used.

The publicKey field points at the public key blob provided by the client.

The public key checking will have either one or two steps. First, if the hasSignature field is not set, there is no signature. Only verify the username and publicKey are expected. This step is optional depending on client configuration, and can save from doing costly public key operations with an invalid user. Second, the hasSignature field is set and signature field points to the client signature. Again the username and publicKey should be checked. wolfSSH will automatically check the signature.

Each of the fields has a size value in octets.

```
typedef struct WS_UserAuthData_PublicKey {
    byte* publicKeyType;
    word32 publicKeyTypeSz;
    byte* publicKey;
    word32 publicKeySz;
    byte hasSignature;
    byte* signature;
    word32 signatureSz;
} WS_UserAuthData_PublicKey;
```

6 Callback Function Setup API

The following functions are used to set up the user authentication callback function.

6.1 Setting the User Authentication Callback Function

```
void wolfSSH_SetUserAuth(WOLFSSH_CTX* ctx , WS_CallbackUserAuth
cb );
```

The callback function is set on the wolfSSL CTX object that is used to create the wolfSSH session objects. All sessions using this CTX will use the same callback function. This context is not to be confused with the callback function's context.

6.2 Setting the User Authentication Callback Context Data

```
void wolfSSH_SetUserAuthCtx(WOLFSSH* ssh , void* ctx );
```

Each wolfSSH session may have its own user authentication context data or share some. The wolfSSH library knows nothing of the contents of this context data. It is up to the application to create, release, and if needed provide a mutex for the data. The callback receives this context data from the library.

6.3 Getting the User Authentication Callback Context Data

```
void* wolfSSH_GetUserAuthCtx(WOLFSSH* ssh );
```

This returns the pointer to the user authentication context data stored in the provided wolfSSH session. This is not to be confused with the wolfSSH's context data used to create the session.

6.4 Setting the Keyboard Authentication Prompts Callback Function

```
void wolfSSH_SetKeyboardAuthPrompts(WOLFSSH_CTX* ctx,
   WS_CallbackKeyboardAuthPrompts cb);
```

The server needs to specify the prompts that are to be given to the client so that it can authenticate in Keyboard-Interactive mode. This callback allows the server to set the prompts ready to send to the client.

Without this set, Keyboard-Interactive mode will be disabled on the server, even if attempts are made to explicitly enable it.

6.5 Example Echo Server User Authentication

The example echo server implements the authentication callback with sample users using passwords and public keys. The example callback, wsUserAuth, is set on the wolfSSH context:

```
wolfSSH_SetUserAuth(ctx, wsUserAuth);
```

The example password file (passwd.txt) is a simple list of usernames and passwords seperated with a colon respectively. The defaults that exist within this file are as follows.

```
jill:upthehill
jack:fetchapail
```

The public key file are the concatenation of the public key outputs of running ssh-keygen twice.

```
ssh-rsa AAAAB3NzaC1yc...d+JI8wrAhfE4x hansel
ssh-rsa AAAAB3NzaC1yc...UoGCPIKuqcFMf gretel
```

All users' authorization data is stored in a linked list of pairs of usernames and SHA-256 hashes of either the password or the public key blob.

The public key blobs in the configuration file are Base64 encoded and are decoded before hashing. The pointer to the list of username-hash pairs is stored into a new wolfSSH session:

```
wolfSSH_SetUserAuthCtx(ssh, &pwMapList);
```

The callback function first checks if the authType is either public key or a password, and returns the general user authentication failure error code if neither. Then it hashes the public key or password passed in via the authData. It then walks through the list trying to find the username, and if not found returns the invalid user error code. If found, it compares the calculated hash of the public key or password passed in and the hash stored in the pair. If they match, the function returns success, otherwise it returns the invalid password or public key error code.

7 Building and Using wolfSSH SFTP

7.1 Building wolfSSH SFTP

It is assumed that wolfSSL has already been built to be used with wolfSSH. To see building instructions for wolfSSL visit Chapter 2.

To build wolfSSH with support for SFTP use –enable-sftp, in the case of building with autotools, or define the macro WOLFSSH_SFTP if building without autotools. An example of this would be:

```
./configure --enable-sftp && make
```

By default the internal buffer size for handling reads and writes for get and put commands is set to 1024 bytes. This value can be overwritten in the case that the application needs to consume less resources or in the case that a larger buffer is desired. To override the default size define the macro WOLFSSH_MAX_SFTP_RW at compile time. An example of setting it would be as follows:

```
./configure --enable-sftp
C_EXTRA_FLAGS'=WOLFSSH_MAX_SFTP_RW=2048
```

7.2 Using wolfSSH SFTP Apps

A SFTP server and client application are bundled with wolfSSH. Both applications get built by autotools when building the wolfSSH library with SFTP support. The server application is located in examples/echoserver/ and is called echoserver. The client application is located in wolfsftp/client/ and is called wolfsftp.

An example of starting up a server that would handle incoming SFTP client connections would be as follow:

```
./examples/echoserver/echoserver
```

Where the command is being ran from the root wolfSSH directory. This starts up a server that is able to handle both SSH and SFTP connections.

Starting the client with specific username:

```
$ ./wolfsftp/client/wolfsftp -u <username>
```

The default “username:password” to run the test is either: “jack:fetchapail” or “jill:upthehill”. The default port is 22222.

A full list of supported commands can be seen with typeing “help” after a connection.

```
wolfSSH sftp> help

Commands :
    cd  <string>                    change directory
    chmod <mode> <path>             change mode
    get <remote file> <local file>  pulls file(s) from server
    ls                              list current directory
    mkdir <dir name>                creates new directory on server
    put <local file> <remote file>  push file(s) to server
    pwd                             list current path
    quit                            exit
    rename <old> <new>              renames remote file
    reget <remote file> <local file> resume pulling file
    reput <remote file> <local file> resume pushing file
    <crtl + c>                      interrupt get/put cmd
```

An example of connecting to another system would be

```
src/wolfssh$ ./examples/sftpclient/wolfsftp -p 22 -u user -h 192.168.1.111
```

8 Port Forwarding

8.1 Building wolfSSH with Port Forwarding

It is assumed that wolfSSL has already been built to be used with wolfSSH. To see building instructions for wolfSSL view Chapter 2.

To build wolfSSH with support for port forwarding use –enable-fwd, in the case of building with autotools, or define the macro `WOLFSSH_FWD` if building without autotools. An example of this would be

```
./configure --enable-fwd && make
```

8.2 Using wolfSSH Port Forwarding Example App

The portfwd example tool will create a “direct-tcpip” style channel. These directions assume you have OpenSSH’s server running in the background with port forwarding enabled. This example forwards the port for the wolfSSL client to the server as the application. It assumes that all programs are run on the same machine in different terminals.

```
src/wolfssl$ ./examples/server/server
src/wolfssh$ ./examples/portfwd/portfwd -p 22 -u <username> \
             -f 12345 -t 11111
src/wolfssl$ ./examples/client/client -p 12345
```

By default, the wolfSSL server listens on port 11111. The client is set to try to connect to port 12345. The portfwd logs in as user “username”, opens a listener on port 12345 and connects to the server on port 11111. Packets are routed back and forth between the client and server. “Hello, wolfSSL!”

The source for portfwd provides an example on how to set up and use the port forwarding support in wolfSSH.

The echoserver will handle local and remote port forwarding. To connect with the ssh tool, using one of the following command lines. You can run either of the ssh command lines from anywhere:

```
src/wolfssl$ ./examples/server/server
src/wolfssh$ ./examples/echoserver/echoserver
anywhere 1$ ssh -p 22222 -L 12345:localhost:11111 jill@localhost
anywhere 2$ ssh -p 22222 -R 12345:localhost:11111 jill@localhost
src/wolfssl$ ./examples/client/client -p 12345
```

This will allow port forwarding between the wolfSSL client and server like in the previous example.

9 Notes and Limitations

In portions of the implementation file attributes are not being considered and default attributes or mode values are used. Specifically in `wolfSSH_SFTP_Open`, getting timestamps from files, and all extended file attributes.

10 Licensing

10.1 Open Source

wolfSSL, yaSSL, wolfCrypt, yaSSH and TaoCrypt software are free software downloads and may be modified to the needs of the user as long as the user adheres to version two of the GPL License. The GPLv2 license can be found on the gnu.org website: http://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

wolfSSH software is a free software download and may be modified to the needs of the user as long as the user adheres to version three of the GPL license. The GPLv3 license can be found on the gnu.org website (https://www.gnu.org/licenses/gpl.html).

10.2 Commercial Licensing

Businesses and enterprises who wish to incorporate wolfSSL products into proprietary appliances or other commercial software products for re-distribution must license commercial versions.

Please contact licensing@wolfssl.com with inquiries.

10.2.1 Support Packages

Support packages for wolfSSL products are available on an annual basis directly from wolfSSL. With three different package options, you can compare them side-by-side and choose the package that best fits your specific needs. Please see our Support Packages page for more details.

11 Support and Consulting

11.1 How to Get Support

For general product support, wolfSSL maintains an online forum for the wolfSSL product family. Please post to the forums or contact wolfSSL directly with any questions.

- wolfSSL Forums: https://www.wolfssl.com/forums
- Email Support: support@wolfssl.com

For information regarding wolfSSL products, questions regarding licensing, or general comments, please contact wolfSSL by emailing **info@wolfssl.com**.

11.1.1 Bugs Reports and Support Issues

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

1. wolfSSL version number
2. Operating System version
3. Compiler version
4. The exact error you are seeing
5. A description of how we can reproduce or try to replicate this problem

With the above information, we will do our best to resolve your problems. Without this information, it is very hard to pinpoint the source of the problem. wolfSSL values your feedback and makes it a top priority to get back to you as soon as possible.

11.2 Consulting

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

11.2.1 Feature Additions and Porting

We can add additional features that you may need which are not currently offered in our products on a contract or co-development basis. We also offer porting services on our products to new host languages or new operating environments.

11.2.2 Competitive Upgrade Program

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

Program Outline:

1. You need to currently be using a commercial competitor to wolfSSL.
2. You will receive up to one week of on-site consulting to switch out your old SSL library for wolfSSL. Travel expenses are not included.
3. Normally, up to one week is the right amount of time for us to make the replacement in your code and do initial testing. Additional consulting on a replacement is available as needed.
4. You will receive the standard wolfSSL royalty free license to ship with your product.

The purpose of this program is to enable users who are currently spending too much on their embedded SSL implementation to move to wolfSSL with ease. If you are interested in learning more, then please contact us at info@wolfssl.com.

11.2.3 Design Consulting

If your application or framework needs to be secured with SSL/TLS but you are uncertain about how the optimal design of a secured system would be structured, we can help!

We offer design consulting for building SSL/TLS security into devices using wolfSSL. Our consultants can provide you with the following services:

12 wolfSSH Updates

12.1 Product Release Information

We regularly post update information on Twitter. For additional release information, you can keep track of our projects on GitHub, follow us on Facebook, or follow our daily blog.

- wolfSSH on GitHub https://www.github.com/wolfssl/wolfssh
- wolfSSL on Twitter https://twitter.com/wolfSSL
- wolfSSL on Facebook https://www.facebook.com/wolfSSL
- wolfSSL on Reddit https://www.reddit.com/r/wolfssl/
- Daily Blog https://wolfssl.com/wolfSSL/Blog/Blog.html

13 API Reference

This section describes the public application program interfaces for the wolfSSH library.

13.1 Error Codes

13.1.1 WS_ErrorCodes (enum)

The following API response codes are defined in: wolfssh/wolfssh/error.h and describe the different types of errors that can occur.

- WS_SUCCESS (0): Function success
- WS_FATAL_ERROR (-1): General function failure
- WS_BAD_ARGUMENT (-2): Function argument out of bounds
- WS_MEMORY_E (-3): Memory allocation error
- WS_BUFFER_E (-4): Input/output buffer size error
- WS_PARSE_E (-5): General parsing error
- WS_NOT_COMPILED (-6): Feature not compiled in
- WS_OVERFLOW_E (-7): Would overflow if continued
- WS_BAD_USAGE (-8): Bad example usage
- WS_SOCKET_ERROR_E (-9): Socket error
- WS_WANT_READ (-10): IO callback would read block error
- WS_WANT_WRITE (-11): IO callback would write block error
- WS_RECV_OVERFLOW_E (-12): Received buffer overflow
- WS_VERSION_E (-13): Peer using wrong version of SSH
- WS_SEND_OOB_READ_E (-14): Attempted to read buffer out of bounds
- WS_INPUT_CASE_E (-15): Bad process input state, programming error
- WS_BAD_FILETYPE_E (-16): Bad filetype
- WS_UNIMPLEMENTED_E (-17): Feature not implemented
- WS_RSA_E (-18): RSA buffer error
- WS_BAD_FILE_E (-19): Bad file
- WS_INVALID_ALGO_ID (-20): invalid algorithm ID
- WS_DECRYPT_E (-21): Decrypt error
- WS_ENCRYPT_E (-22): Encrypt error
- WS_VERIFY_MAC_E (-23): verify mac error
- WS_CREATE_MAC_E (-24): Create mac error
- WS_RESOURCE_E (-25): Insufficient resources for new channel
- WS_INVALID_CHANTYPE (-26): Invalid channel type
- WS_INVALID_CHANID(-27): Peer requested invalid channel ID
- WS_INVALID_USERNAME(-28): Invalid user name
- WS_CRYPTO_FAILED(-29): Crypto action failed
- WS_INVALID_STATE_E(-30): Invalid State
- WC_EOF(-31): End of File
- WS_INVALID_PRIME_CURVE(-32): Invalid prime curve in ECC
- WS_ECC_E(-33): ECDSA buffer error
- WS_CHANOPEN_FAILED(-34): Peer returned channel open failure
- WS_REKEYING(-35): Rekeying with peer
- WS_CHANNEL_CLOSED(-36): Channel closed

13.1.2 WS_IOerrors (enum)

These are the return codes the library expects to receive from a user-provided I/O callback. Otherwise the library expects the number of bytes read or written from the I/O action.

- WS_CBIO_ERR_GENERAL (-1): General unexpected error

- WS_CBIO_ERR_WANT_READ (-2): Socket read would block, call again
- WS_CBIO_ERR_WANT_WRITE (-2): Socket write would block, call again
- WS_CBIO_ERR_CONN_RST (-3): Connection reset
- WS_CBIO_ERR_ISR (-4): Interrupt
- WS_CBIO_ERR_CONN_CLOSE (-5): Connection closed or EPIPE
- WS_CBIO_ERR_TIMEOUT (-6): Socket timeout”

13.2 Initialization / Shutdown

13.2.1 wolfSSH_Init()

Synopsis

Description

Initializes the wolfSSH library for use. Must be called once per application and before any other calls to the library.

Return Values

WS_SUCCESS WS_CRYPTO_FAILED

Parameters

None

See Also

wolfSSH_Cleanup()

```
#include <wolfssh/ssh.h>
int wolfSSH_Init(void);
```

13.2.2 wolfSSH_Cleanup()

Synopsis

Description

Cleans up the wolfSSH library when done. Should be called at before termination of the application. After calling, do not make any more calls to the library.

Return Values

WS_SUCCESS

WS_CRYPTO_FAILED

Parameters

None

See Also

wolfSSH_Init()

```
#include <wolfssh/ssh.h>
int wolfSSH_Cleanup(void);
```

13.3 Debugging output functions

13.3.1 wolfSSH_Debugging_ON()

Synopsis

Description

Enables debug logging during runtime. Does nothing when debugging is disabled at build time.

Return Values

None

Parameters

None

See Also

wolfSSH_Debugging_OFF()

```
#include <wolfssh/ssh.h>
void wolfSSH_Debugging_ON(void);
```

13.3.2 wolfSSH_Debugging_OFF()

Synopsis

Description

Disables debug logging during runtime. Does nothing when debugging is disabled at build time.

Return Values

None

Parameters

None

See Also

wolfSSH_Debugging_ON()

```
#include <wolfssh/ssh.h>
void wolfSSH_Debugging_OFF(void);
```

13.4 Context Functions

13.4.1 wolfSSH_CTX_new()

Synopsis

Description

Creates a wolfSSH context object. This object can be configured and then used as a factory for wolfSSH session objects.

Return Values

WOLFSSH_CTX* – returns pointer to allocated WOLFSSH_CTX object or NULL

Parameters

side – indicate client side (unimplemented) or server side **heap** – pointer to a heap to use for memory allocations

See Also

wolfSSH_wolfSSH_CTX_free()

```
#include <wolfssh/ssh.h>
WOLFSSH_CTX* wolfSSH_CTX_new(byte side , void* heap );
```

13.4.2 wolfSSH_CTX_free()

Synopsis

Description

Deallocates a wolfSSH context object.

Return Values

None

Parameters

ctx – the wolfSSH context used to initialize the wolfSSH session

See Also

wolfSSH_wolfSSH_CTX_new()

```
#include <wolfssh/ssh.h>
void wolfSSH_CTX_free(WOLFSSH_CTX* ctx );
```

13.4.3 wolfSSH_CTX_SetBanner()

Synopsis

Description

Sets a banner message that a user can see.

Return Values

WS_BAD_ARGUMENT WS_SUCCESS

Parameters

ssh - Pointer to wolfSSH session **newBanner** - The banner message text.

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetBanner(WOLFSSH_CTX* ctx , const char*
newBanner );
```

13.4.4 wolfSSH_CTX_UsePrivateKey_buffer()

Synopsis

Description

This function loads a private key buffer into the SSH context. It is called with a buffer as input instead of a file. The buffer is provided by the **in** argument of size **inSz**. The argument **format** specifies the type of buffer: **WOLFSSH_FORMAT_ASN1** or **WOLFSSL_FORMAT_PEM** (unimplemented at this time).

Return Values

WS_SUCCESS WS_BAD_ARGUMENT – at least one of the parameters is invalid **WS_BAD_FILETYPE_E** – wrong format **WS_UNIMPLEMENTED_E** – support for PEM format not implemented **WS_MEMORY_E** – out of memory condition **WS_RSA_E** – cannot decode RSA key **WS_BAD_FILE_E** – cannot parse buffer

Parameters

ctx – pointer to the wolfSSH context **in** – buffer containing the private key to be loaded **inSz** – size of the input buffer **format** – format of the private key located in the input buffer

See Also

wolfSSH_UseCert_buffer() wolfSSH_UseCaCert_buffer()

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_UsePrivateKey_buffer(WOLFSSH_CTX* ctx ,
const byte* in , word32 inSz , int format );
```

13.5 SSH Session Functions

13.5.1 wolfSSH_new()

Synopsis

Description

Creates a wolfSSH session object. It is initialized with the provided wolfSSH context.

Return Values

WOLFSSH* – returns pointer to allocated WOLFSSH object or NULL

Parameters

ctx – the wolfSSH context used to initialize the wolfSSH session

See Also

wolfSSH_free()

```
#include <wolfssh/ssh.h>
WOLFSSH* wolfSSH_new(WOLFSSH_CTX* ctx );
```

13.5.2 wolfSSH_free()

Synopsis

Description

Deallocates a wolfSSH session object.

Return Values

None

Parameters

ssh – session to deallocate

See Also

wolfSSH_new()

```
#include <wolfssh/ssh.h>
void wolfSSH_free(WOLFSSH* ssh );
```

13.5.3 wolfSSH_set_fd()

Synopsis

Description

Assigns the provided file descriptor to the ssh object. The ssh session will use the file descriptor for network I/O in the default I/O callbacks.

Return Values

13.5.3.1 WS_SUCCESS WS_BAD_ARGUMENT – one of the parameters is invalid

Parameters

ssh – session to set the fd **fd** – file descriptor for the socket used by the session

See Also

wolfSSH_get_fd()

```
#include <wolfssh/ssh.h>
int wolfSSH_set_fd(WOLFSSH* ssh , int fd );
```

13.5.4 wolfSSH_get_fd()

Synopsis

Description

This function returns the file descriptor (**fd**) used as the input/output facility for the SSH connection. Typically this will be a socket file descriptor.

Return Values

int – file descriptor **WS_BAD_ARGUEMENT**

Parameters

ssh – pointer to the SSL session.

See Also

wolfSSH_set_fd()

```
#include <wolfssh/ssh.h>
int wolfSSH_get_fd(const WOLFSSH* ssh );
```

13.6 Data High Water Mark Functions

13.6.1 wolfSSH_SetHighwater()

Synopsis

Description

Sets the highwater mark for the ssh session.

Return Values

WS_SUCCESS WS_BAD_ARGUMENT

Parameters

ssh - Pointer to wolfSSH session **highwater** - data indicating the highwater security mark

```
#include <wolfssh/ssh.h>
int wolfSSH_SetHighwater(WOLFSSH* ssh , word32 highwater );
```

13.6.2 wolfSSH_GetHighwater()

Synopsis

Description

Returns the highwater security mark

Return Values

word32 - The highwater security mark.

Parameters

ssh - Pointer to wolfSSH session

```
#include <wolfssh/ssh.h>
word32 wolfSSH_GetHighwater(WOLFSSH* ssh );
```

13.6.3 wolfSSH_SetHighwaterCb()

Synopsis

Description

The wolfSSH_SetHighwaterCb function sets the highwater security mark for the SSH session as well as the high water call back.

Return Values

none

Parameters

ctx – The wolfSSH context used to initialize the wolfSSH session. **highwater** - The highwater security mark. **cb** - The call back highwater function.

```
#include <wolfssh/ssh.h>
void wolfSSH_SetHighwaterCb(WOLFSSH_CTX* ctx , word32 highwater ,
WS_CallbackHighwater cb );
```

13.6.4 wolfSSH_SetHighwaterCtx()

Synopsis

Description

The wolfSSH_SetHighwaterCTX function sets the highwater security mark for the given context.

Return Values

none

Parameters

ssh - pointer to wolfSSH session **ctx** - pointer to highwater security mark in the wolfSSH context.

```
#include <wolfssh/ssh.h>
void wolfSSH_SetHighwaterCtx(WOLFSSH* ssh, void* ctx);
```

13.6.5 wolfSSH_GetHighwaterCtx()

Synopsis

Description

The wolfSSH_GetHighwaterCtx() returns the highwaterCtx security mark from the SSH session.

Return Values

void* - the highwater security mark **NULL** - if there is an error with the WOLFSSH object.

Parameters

ssh - pointer to WOLFSSH object

```
#include <wolfssh/ssh.h>
void wolfSSH_GetHighwaterCtx(WOLFSSH* ssh );
```

13.7 Error Checking

13.7.1 wolfSSH_get_error()

Synopsis

Description

Returns the error set in the wolfSSH session object.

Return Values

WS_ErrorCodes (enum)

Parameters

ssh – pointer to WOLFSSH object

See Also

wolfSSH_get_error_name()

```
#include <wolfssh/ssh.h>
int wolfSSH_get_error(const WOLFSSH* ssh );
```

13.7.2 wolfSSH_get_error_name()

Synopsis

Description

Returns the name of the error set in the wolfSSH session object.

Return Values

const char* – error name string

Parameters

ssh – pointer to WOLFSSH object

See Also

wolfSSH_get_error()

```
#include <wolfssh/ssh.h>
const char* wolfSSH_get_error_name(const WOLFSSH* ssh );
```

13.7.3 wolfSSH_ErrorToName()

Synopsis

Description

Returns the name of an error when called with an error number in the parameter.

Return Values

const char* – name of error string

Parameters

err - the int value of the error

```
#include <wolfssh/ssh.h>
const char* wolfSSH_ErrorToName(int err );
```

13.8 I/O Callbacks

13.8.1 wolfSSH_SetIORecv()

Synopsis

Description

This function registers a receive callback for wolfSSL to get input data.

Return Values

None

Parameters

ctx – pointer to the SSH context **cb** – function to be registered as the receive callback for the wolfSSH context, **ctx**. The signature of this function must follow that as shown above in the Synopsis section.

```
#include <wolfssh/ssh.h>
void wolfSSH_SetIORecv(WOLFSSH_CTX* ctx , WS_CallbackIORecv cb );
```

13.8.2 wolfSSH_SetIOSend()

Synopsis

Description

This function registers a send callback for wolfSSL to write output data.

Return Values

None

Parameters

ctx – pointer to the wolfSSH context **cb** – function to be registered as the send callback for the wolfSSH context, **ctx**. The signature of this function must follow that as shown above in the Synopsis section.

```
#include <wolfssh/ssh.h>
void wolfSSH_SetIOSend(WOLFSSH_CTX* ctx , WS_CallbackIOSend cb );
```

13.8.3 wolfSSH_SetIOReadCtx()

Synopsis

Description

This function registers a context for the SSH session receive callback function.

Return Values

None

Parameters

ssh – pointer to WOLFSSH object **ctx** – pointer to the context to be registered with the SSH session (**ssh**) receive callback function.

```
#include <wolfssh/ssh.h>
void wolfSSH_SetIOReadCtx(WOLFSSH* ssh , void* ctx );
```

13.8.4 wolfSSH_SetIOWriteCtx()

Synopsis

Description

This function registers a context for the SSH session's send callback function.

Return Values

None

Parameters

ssh – pointer to WOLFSSH session. **ctx** – pointer to be registered with the SSH session's (**ssh**) send callback function.

```
#include <wolfssh/ssh.h>
void wolfSSH_SetIOWriteCtx(WOLFSSH* ssh , void* ctx );
```

13.8.5 wolfSSH_GetIOReadCtx()

Synopsis

Description

This function return the ioReadCtx member of the WOLFSSH structure.

Return Values

Void* - pointer to the ioReadCtx member of the WOLFSSH structure.

Parameters

ssh – pointer to WOLFSSH object

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetIOReadCtx(WOLFSSH* ssh );
```

13.8.6 wolfSSH_GetIOWriteCtx()

Synopsis

Description

This function returns the ioWriteCtx member of the WOLFSSH structure.

Return Values

Void* – pointer to the ioWriteCtx member of the WOLFSSH structure.

Parameters

ssh – pointer to WOLFSSH object

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetIOWriteCtx(WOLFSSH* ssh );
```

13.9 User Authentication

13.9.1 wolfSSH_SetUserAuth()

Synopsis

Description

The wolfSSH_SetUserAuth() function is used to set the user authentication for the current wolfSSH context if the context does not equal NULL.

Return Values

None

Parameters

ctx – pointer to the wolfSSH context **cb** – call back function for the user authentication

```
#include <wolfssh/ssh.h>
void wolfSSH_SetUserAuth(WOLFSSH_CTX* ctx ,
WS_CallbackUserAuth cb )
```

13.9.2 wolfSSH_SetUserAuthCtx()

Synopsis

Description

The wolfSSH_SetUserAuthCtx() function is used to set the value of the user authentication context in the SSH session.

Return Values

None

Parameters

ssh – pointer to WOLFSSH object **userAuthCtx** – pointer to the user authentication context

```
#include <wolfssh/ssh.h>
void wolfSSH_SetUserAuthCtx(WOLFSSH* ssh , void*
userAuthCtx )
```

13.9.3 wolfSSH_GetUserAuthCtx()

Synopsis

Description

The wolfSSH_GetUserAuthCtx() function is used to return the pointer to the user authentication context.

Return Values

Void* – pointer to the user authentication context **Null** – returns if ssh is equal to NULL

Parameters

ssh – pointer to WOLFSSH object

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetUserAuthCtx(WOLFSSH* ssh )
```

13.9.4 wolfSSH_SetKeyboardAuthPrompts()

Synopsis

Description

The wolfSSH_SetKeyboardAuthPrompts() function is used to setup the callback which will provide the server with the prompts to send to the client.

Return Values

None

Parameters

ctx - pointer to the wolfSSH context **cb** - callback function to provide the keyboard prompts

```
#include <wolfssh/ssh.h>
void wolfSSH_SetKeyboardAuthPrompts(WOLFSSH_CTX* ctx,
                                    WS_CallbackKeyboardAuthPrompts cb)
```

13.9.5 wolfSSH_SetKeyboardAuthCtx()

Synopsis

Description

The wolfSSH_SetKeyboardAuthCtx() function is used to setup the user context for the wolfSSH_SetKeyboardAuthPrompts() function.

Return Values

None

Parameters

ssh - pointer to the WOLFSSH object **keyboardAuthCtx* - pointer to the user context data

```
#include <wolfssh/ssh.h>
void wolfSSH_SetKeyboardAuthCtx(WOLFSSH* ssh, void* keyboardAuthCtx)
```

13.10 Set Username

13.10.1 wolfSSH_SetUsername()

Synopsis

Description

Sets the username required for the SSH connection.

Return Values

WS_BAD_ARGUMENT WS_SUCCESS WS_MEMORY_E

Parameters

ssh - Pointer to wolfSSH session **username** - The input username for the SSH connection.

```
#include <wolfssh/ssh.h>
int wolfSSH_setUsername(WOLFSSH* ssh , const char* username );
```

13.11 Connection Functions

13.11.1 wolfSSH_accept()

Synopsis

Description

wolfSSH_accept is called on the server side and waits for an SSH client to initiate the SSH handshake.

wolfSSL_accept() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSH_accept() will return when the underlying I/O could not satisfy the needs of wolfSSH_accept to continue the handshake. In this case, a call to wolfSSH_get_error() will yield either **WS_WANT_READ** or **WS_WANT_WRITE**. The calling process must then repeat the call to wolfSSH_accept when data is available to read and wolfSSH will pick up where it left off. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSH_accept() will only return once the handshake has been finished or an error occurred.

Return Values

WS_SUCCESS - The function succeeded. **WS_BAD_ARGUMENT** - A parameter value was null. **WS_FATAL_ERROR** – There was an error, call wolfSSH_get_error() for more detail

Parameters

ssh – pointer to the wolfSSH session

See Also

wolfSSH_stream_read()

```
#include <wolfssh/ssh.h>
int wolfSSH_accept(WOLFSSH* ssh);
```

13.11.2 wolfSSH_connect()

Synopsis

Description

This function is called on the client side and initiates an SSH handshake with a server. When this function is called, the underlying communication channel has already been set up.

wolfSSH_connect() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSH_connect() will return when the underlying I/O could not satisfy the needs of wolfSSH_connect to continue the handshake. In this case, a call to wolfSSH_get_error() will yield either **WS_WANT_READ** or **WS_WANT_WRITE**. The calling process must then repeat the call to wolfSSH_connect() when the underlying I/O is ready and wolfSSH will pick up where it left off. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSH_connect() will only return once the handshake has been finished or an error occurred.

Return Values

WS_BAD_ARGUMENT WS_FATAL_ERROR WS_SUCCESS - This will return if the call is successful.

Parameters

ssh - Pointer to wolfSSH session

```
#include <wolfssh/ssh.h>
int wolfSSH_connect(WOLFSSH* ssh);
```

13.11.3 wolfSSH_shutdown()

Synopsis

Description

Closes and disconnects the SSH channel.

Return Values

WS_BAD_ARGUMENT - returned if the parameter is NULL **WS_SUCCES** - returns when everything has been correctly shutdown

Parameters

ssh - Pointer to wolfSSH session

```
#include <wolfssh/ssh.h>
int wolfSSH_shutdown(WOLFSSH* ssh);
```

13.11.4 wolfSSH_stream_read()

Synopsis

Description

wolfSSH_stream_read reads up to **bufSz** bytes from the internal decrypted data stream buffer. The bytes are removed from the internal buffer.

wolfSSH_stream_read() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSH_stream_read() will return when the underlying I/O could not satisfy the needs of wolfSSH_stream_read to continue the read. In this case, a call to wolfSSH_get_error() will yield

either **WS_WANT_READ** or **WS_WANT_WRITE**. The calling process must then repeat the call to wolfSSH_stream_read when data is available to read and wolfSSH will pick up where it left off. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSH_stream_read() will only return when data is available or an error occurred.

Return Values

>0 – number of bytes read upon success **0** – returned on socket failure caused by either a clean connection shutdown or a socket. **WS_BAD_ARGUMENT** – returns if one or more parameters is equal to NULL **WS_EOF** – returns when end of stream is reached **WS_FATAL_ERROR** – there was an error, call **wolfSSH_get_error()** for more detail **WS_REKEYING** if currently a rekey is in process, use wolfSSH_worker() to complete

Parameters

ssh – pointer to the wolfSSH session

```
#include <wolfssh/ssh.h>
int wolfSSH_stream_read(WOLFSSH* ssh ,
byte* buf , word32 bufSz );
```

buf – buffer where wolfSSH_stream_read() will place the data **bufSz** – size of the buffer

See Also

wolfSSH_accept() wolfSSH_stream_send()

13.11.5 wolfSSH_stream_send()

Synopsis

Description

wolfSSH_stream_send writes **bufSz** bytes from buf to the SSH stream data buffer. wolfSSH_stream_send() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSH_stream_send() will return a want write error when the underlying I/O could not satisfy the needs of wolfSSH_stream_send and there is still pending data in the SSH stream data buffer to be sent. In this case, a call to wolfSSH_get_error() will yield either **WS_WANT_READ** or **WS_WANT_WRITE**. The calling process must then repeat the call to wolfSSH_stream_send when the socket is ready to send and wolfSSH will send out any pending data left in the SSH stream data buffer then pull data from the input **buf**. When using a non-blocking socket, nothing needs to be done, but select() can be used to check for the required condition.

If the underlying I/O is blocking, wolfSSH_stream_send() will only return when the data has been sent or an error occurred.

In cases where I/O want write/read is not the error encountered (i.e. WS_REKEYING) then wolfSSH_worker() should be called until the internal SSH processes are completed.

Return Values

>0 – number of bytes written to SSH stream data buffer upon success **0** – returned on socket failure caused by either a clean connection shutdown or a socket error, call **wolfSSH_get_error()** for more detail **WS_FATAL_ERROR** – there was an error, call wolfSSH_get_error() for more detail **WS_BAD_ARGUMENT** if any of the parameters is null **WS_REKEYING** if currently a rekey is in process, use wolfSSH_worker() to complete

Parameters

ssh – pointer to the wolfSSH session **buf** – buffer wolfSSH_stream_send() will send

```
#include <wolfssh/ssh.h>
int wolfSSH_stream_send(WOLFSSH* ssh , byte* buf , word32
bufSz );
```

bufSz – size of the buffer

See Also

wolfSSH_accept() wolfSSH_stream_read()

13.11.6 wolfSSH_stream_exit()

Synopsis

Description

This function is used to exit the SSH stream.

Return Values

WS_BAD_ARGUMENT - returned if a parameter value is NULL **WS_SUCCESS** - returns if function was a success

Parameters

ssh – Pointer to wolfSSH session **status** – the status of the SSH connection

```
#include <wolfssh/ssh.h>
int wolfSSH_stream_exit(WOLFSSH* ssh, int status);
```

13.11.7 wolfSSH_TriggerKeyExchange()

Synopsis

Description

Triggers key exchange process. Prepares and sends packet of allocated handshake info.

Return Values

WS_BAD_ARGUEMENT – if **ssh** is NULL **WS_SUCCESS**

Parameters

ssh – pointer to the wolfSSH session

```
#include <wolfssh/ssh.h>
int wolfSSH_TriggerKeyExchange(WOLFSSH* ssh );
```

13.12 Channel Callbacks

Interfaces to the wolfSSH library return single int values. Communicating status of asynchronous information, like the peer opening a channel, isn't easy with that interface. wolfSSH uses callback functions to notify the calling application of changes in state of a channel.

There are callback functions for receipt of the following SSHv2 protocol messages:

- SSH_MSG_CHANNEL_OPEN
- SSH_MSG_CHANNEL_OPEN_CONFIRMATION
- SSH_MSG_CHANNEL_OPEN_FAILURE

- SSH_MSG_CHANNEL_REQUEST
  - "shell"
  - "subsystem"
  - "exec"
- SSH_MSG_CHANNEL_EOF
- SSH_MSG_CHANNEL_CLOSE

13.12.1 Callback Function Prototypes

The channel callback functions all take a pointer to a **WOLFSSH_CHANNEL** object, *channel*, and a pointer to the application defined data structure, *ctx*. Properties about the channel may be queried using API functions.

```
typedef int (*WS_CallbackChannelOpen)(WOLFSSH_CHANNEL* channel, void* ctx);
typedef int (*WS_CallbackChannelReq)(WOLFSSH_CHANNEL* channel, void* ctx);
typedef int (*WS_CallbackChannelEof)(WOLFSSH_CHANNEL* channel, void* ctx);
typedef int (*WS_CallbackChannelClose)(WOLFSSH_CHANNEL* channel, void* ctx);
```

13.12.2 wolfSSH_CTX_SetChannelOpenCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelOpenCb(WOLFSSH_CTX* ctx,
        WS_CallbackChannelOpen cb);
```

Description

Sets the callback function, *cb*, into the wolfSSH *ctx* used when a Channel Open (**SSH_MSG_CHANNEL_OPEN**) message is received from the peer.

Return Values

- **WS_SUCCESS** - Setting callback in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.3 wolfSSH_CTX_SetChannelOpenRespCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelOpenRespCb(WOLFSSH_CTX* ctx,
        WS_CallbackChannelOpen confCb,
        WS_CallbackChannelOpen failCb);
```

Description

Sets the callback functions, *confCb* and *failCb*, into the wolfSSH *ctx* used when a Channel Open Confirmation (**SSH_MSG_CHANNEL_OPEN_CONFIRMATION**) or a Channel Open Failure (**SSH_MSG_CHANNEL_OPEN_FAILURE**) message is received from the peer.

Return Values

- **WS_SUCCESS** - Setting callbacks in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.4 wolfSSH_CTX_SetChannelReqShellCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelReqShellCb(WOLFSSH_CTX* ctx,
        WS_CallbackChannelReq cb);
```

Description

Sets the callback function, *cb*, into the wolfSSH *ctx* used when a Channel Request (**SSH_MSG_CHANNEL_REQUEST**) message is received from the peer for a *shell*.

Return Values

- **WS_SUCCESS** - Setting callback in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.5 wolfSSH_CTX_SetChannelReqSubsysCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelReqSubsysCb(WOLFSSH_CTX* ctx,
        WS_CallbackChannelReq cb);
```

Description

Sets the callback function, *cb*, into the wolfSSH *ctx* used when a Channel Request (**SSH_MSG_CHANNEL_REQUEST**) message is received from the peer for a *subsystem*. A common example of a subsystem is SFTP.

Return Values

- **WS_SUCCESS** - Setting callback in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.6 wolfSSH_CTX_SetChannelReqExecCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelReqExecCb(WOLFSSH_CTX* ctx,
        WS_CallbackChannelReq cb);
```

Description

Sets the callback function, *cb*, into the wolfSSH *ctx* used when a Channel Request (**SSH_MSG_CHANNEL_REQUEST**) message is received from the peer for a command to *exec*.

Return Values

- **WS_SUCCESS** - Setting callback in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.7 wolfSSH_CTX_SetChannelEofCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelEof(WOLFSSH_CTX* ctx,
        WS_CallbackChannelEof cb);
```

Description

Sets the callback function, *cb*, into the wolfSSH *ctx* used when a Channel EOF (**SSH_MSG_CHANNEL_EOF**) message is received from the peer. This message indicates that the peer isn't going to transmit any more data on this channel.

Return Values

- **WS_SUCCESS** - Setting callback in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.8 wolfSSH_CTX_SetChannelCloseCb

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_CTX_SetChannelClose(WOLFSSH_CTX* ctx,
        WS_CallbackChannelClose cb);
```

Description

Sets the callback function, *cb*, into the wolfSSH *ctx* used when a Channel Close (**SSH_MSG_CHANNEL_CLOSE**) message is received from the peer. This message indicates that the peer is interested in terminating this channel.

Return Values

- **WS_SUCCESS** - Setting callback in *ctx* was successful
- **WS_SSH_CTX_NULL_E** - *ctx* is **NULL**

13.12.9 wolfSSH_SetChannelOpenCtx

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_SetChannelOpenCtx(WOLFSSH* ssh, void* ctx);
```

Description

Sets the context, *ctx*, into the wolfSSH *ssh* object used when the callback for the Channel Open (**SSH_MSG_CHANNEL_OPEN**) message, Channel Open Confirmation (**SSH_MSG_CHANNEL_CONFIRMATION**) message, or Channel Open Failure (**SSH_MSG_CHANNEL_FAILURE**) is received from the peer.

Return Values

- **WS_SUCCESS** - Setting context in *ssh* was successful
- **WS_SSH_NULL_E** - *ssh* is **NULL**

13.12.10 wolfSSH_SetChannelReqCtx

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_SetChannelReqCtx(WOLFSSH* ssh, void* ctx);
```

Description

Sets the context, *ctx*, into the wolfSSH *ssh* object used when the callback for the Channel Request (**SSH_MSG_CHANNEL_REQUEST**) message is received from the peer.

Return Values

- **WS_SUCCESS** - Setting context in *ssh* was successful
- **WS_SSH_NULL_E** - *ssh* is **NULL**

13.12.11 wolfSSH_SetChannelEofCtx

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_SetChannelEofCtx(WOLFSSH* ssh, void* ctx);
```

Description

Sets the context, *ctx*, into the wolfSSH *ssh* object used when the callback for the Channel EOF (**SSH_MSG_CHANNEL_EOF**) message is received from the peer.

Return Values

- **WS_SUCCESS** - Setting context in *ssh* was successful
- **WS_SSH_NULL_E** - *ssh* is **NULL**

13.12.12 wolfSSH_SetChannelCloseCtx

Synopsis

```
#include <wolfssh/ssh.h>
int wolfSSH_SetChannelCloseCtx(WOLFSSH* ssh, void* ctx);
```

Description

Sets the context, *ctx*, into the wolfSSH *ssh* object used when the callback for the Channel Close (**SSH_MSG_CHANNEL_CLOSE**) message is received from the peer.

Return Values

- **WS_SUCCESS** - Setting context in *ssh* was successful
- **WS_SSH_NULL_E** - *ssh* is **NULL**

13.12.13 wolfSSH_GetChannelOpenCtx

Synopsis

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetChannelOpenCtx(WOLFSSH* ssh);
```

Description

Gets the context from the wolfSSH *ssh* object used when the callback for the Channel Open (**SSH_MSG_CHANNEL_OPEN**) message.

Return Values

- pointer to the context data

13.12.14 wolfSSH_GetChannelReqCtx

Synopsis

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetChannelReqCtx(WOLFSSH* ssh);
```

Description

Gets the context from the wolfSSH *ssh* object used when the callback for the Channel Request (**SSH_MSG_CHANNEL_REQUEST**) message.

Return Values

- pointer to the context data

13.12.15 wolfSSH_GetChannelEofCtx

Synopsis

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetChannelEofCtx(WOLFSSH* ssh);
```

Description

Gets the context from the wolfSSH *ssh* object used when the callback for the Channel EOF (**SSH_MSG_CHANNEL_EOF**) message.

Return Values

- pointer to the context data

13.12.16 wolfSSH_GetChannelCloseCtx

Synopsis

```
#include <wolfssh/ssh.h>
void* wolfSSH_GetChannelCloseCtx(WOLFSSH* ssh);
```

Description

Gets the context from the wolfSSH *ssh* object used when the callback for the Channel Close (**SSH_MSG_CHANNEL_CLOSE**) message.

Return Values

- pointer to the context data

13.12.17 wolfSSH_ChannelGetSessionType

Synopsis

```
#include <wolfssh/ssh.h>
WS_SessionType wolfSSH_ChannelGetSessionType(const WOLFSSH_CHANNEL* channel);
```

Description

Returns the **WS_SessionType** for the specified *channel*.

Return Values

- **WS_SessionType** - type for the session

13.12.18 wolfSSH_ChannelGetSessionCommand

Synopsis

```
#include <wolfssh/ssh.h>
const char* wolfSSH_ChannelGetSessionCommand(const WOLFSSH_CHANNEL* channel);
```

Description

Returns a pointer to the command the user wishes to execute over the specified *channel*.

Return Values

- **const char*** - pointer to the string holding the command sent by the user

13.13 Testing Functions

13.13.1 wolfSSH_GetStats()

Synopsis

Description

Updates **txCount** , **rxCount** , **seq** , and **peerSeq** with their respective **ssh** session statistics.

Return Values

none

Parameters

ssh – pointer to the wolfSSH session **txCount** – address where total transferred bytes in **ssh** session are stored. **rxCount** – address where total received bytes in **ssh** session are stored. **seq** – packet sequence number is initially 0 and is incremented after every packet **peerSeq** – peer packet sequence number is initially 0 and is incremented after every packet

```
#include <wolfssh/ssh.h>
void wolfSSH_GetStats(WOLFSSH* ssh , word32* txCount , word32*
rxCount ,
word32* seq , word32* peerSeq )
```

13.13.2 wolfSSH_KDF()

Synopsis

Description

This is used so that the API test can do known answer tests for the key derivation.

The Key Derivation Function derives a symmetric **key** based on source keying material, **k** and **h**. Where **k** is the Diffie-Hellman shared secret and **h** is the hash of the handshake that was produced during initial key exchange. Multiple types of keys could be derived which are specified by the **keyId** and **hashId**.

```
Initial IV client to server: keyId = A
Initial IV server to client: keyId = B
Encryption key client to server: keyId = C
Encryption key server to client: keyId = D
Integrity key client to server: keyId = E
Integrity key server to client : keyId = F
```

Return Values

WS_SUCCESS WS_CRYPTO_FAILED

Parameters

hashId – type of hash to generate keying material. e.g. (WC_HASH_TYPE_SHA and WC_HASH_TYPE_SHA256) **keyId** – letter A - F to indicate which key to make **key** – generated key used for comparisons to expected key

```
#include <wolfssh/ssh.h>
int wolfSSH_KDF(byte hashId , byte keyId , byte* key , word32
keySz ,
const byte* k , word32 kSz , const byte* h , word32
hSz ,
const byte* sessionId , word32 sessionIdSz );
```

keySz – needed size of **key k** – shared secret from the Diffie-Hellman key exchange **kSz** – size of the shared secret (**k**) **h** – hash of the handshake that was produced during key exchange **hSz** – size of the hash (**h**) **sessionId** – unique identifier from first **h** calculated. **sessionIdSz** – size of the **sessionId**

13.14 Session Functions

13.14.1 wolfSSH_GetSessionType()

Synopsis

Description

The wolfSSH_GetSessionType() is used to return the type of session

Return Values

WOLFSSH_SESSION_UNKNOWN WOLFSSH_SESSION_SHELL WOLFSSH_SESSION_EXEC WOLFSSH_SESSION_SUBSYSTEM

Parameters

ssh - pointer to wolfSSH session

```
#include <wolfssh/ssh.h>
WS_SessionType wolfSSH_GetSessionType(const WOLFSSH* ssh );
```

13.14.2 wolfSSH_GetSessionCommand()

Synopsis

Description

This function is used to return the current command in the session.

Return Values

const char* - Pointer to command

Parameters

ssh - pointer to wolfSSH session

```
#include <wolfssh/ssh.h>
const char* wolfSSH_GetSessionCommand(const WOLFSSH* ssh );
```

13.15 Port Forwarding Functions

13.15.1 wolfSSH_ChannelFwdNew()

Synopsis

Description

Sets up a TCP/IP forwarding channel on a WOLFSSH session. When the SSH session is connected and authenticated, a local listener is created on the interface for address *host* on port *hostPort*. Any new connections on that listener will trigger a new channel request to the SSH server to establish a connection to *host* on port *hostPort*.

Return Values

WOLFSSH_CHAN* – NULL on error or new channel record

Parameters

ssh – wolfSSH session **host** – host address to bind listener **hostPort** – host port to bind listener **origin** – IP address of the originating connection **originPort** – port number of the originating connection

```
#include <wolfssh/ssh.h>
WOLFSSH_CHANNEL* wolfSSH_ChannelFwdNew(WOLFSSH* ssh ,
const char* host , word32 hostPort ,
const char* origin , word32 originPort );
```

13.15.2 wolfSSH_ChannelFree()

Synopsis

Description

Releases the memory allocated for the channel *channel*. The channel is removed from its session's channel list.

Return Values

int – error code

Parameters

channel – wolfSSH channel to free

```
#include <wolfssh/ssh.h>
int wolfSSH_ChannelFree(WOLFSSH_CHANNEL* channel );
```

13.15.3 wolfSSH_worker()

Synopsis

Description

The wolfSSH worker function babysits the connection and as data is received processes it. SSH sessions have many bookkeeping messages for the session and this takes care of them automatically. When data for a particular channel is received, the worker places the data into the channel. (The function wolfSSH_stream_read() does much the same but also returns the receive data for a single channel.) wolfSSH_worker() will perform the following actions:

1. Attempt to send any pending data in the *outputBuffer*.
2. Call *DoReceive()* on the session's socket.
3. If data is received for a particular channel, return data received notice and set the channel ID.

Return Values

int – error or status **WS_CHANNEL_RXD** – data has been received on a channel and the ID is set

Parameters

ssh – pointer to the wolfSSH session **id** – pointer to the location to save the ID value

```
#include <wolfssh/ssh.h>
int wolfSSH_worker(WOLFSSH* ssh , word32* channelId );
```

13.15.4 wolfSSH_ChannelGetId()

Synopsis

Description

Given a channel, returns the ID or peer's ID for the channel.

Return Values

int – error code

Parameters

channel – pointer to channel **id** – pointer to location to save the ID value **peer** – either self (my channel ID) or peer (my peer's channel ID)

```
#include <wolfssh/ssh.h>
int wolfSSH_ChannelGetId(WOLFSSH_CHANNEL* channel ,
word32* id , byte peer );
```

13.15.5 wolfSSH_ChannelFind()

Synopsis

Description

Given a session *ssh* , find the channel associated with *id*.

Return Values

WOLFSSH_CHANNEL* – pointer to the channel, NULL if the ID isn't in the list

Parameters

ssh – wolfSSH session **id** – channel ID to find **peer** – either self (my channel ID) or peer (my peer's channel ID)

```
#include <wolfssh/ssh.h>
WOLFSSH_CHANNEL* wolfSSH_ChannelFind(WOLFSSH* ssh ,
word32 id , byte peer );
```

13.15.6 wolfSSH_ChannelRead()

Synopsis

Description

Copies data out of a channel object.

Return Values

int – bytes read **>0** – number of bytes read upon success **0** – returns on socket failure cause by either a clean connection shutdown or a socket error, call wolfSSH_get_error() for more detail **WS_FATAL_ERROR** – there was some other error, call wolfSSH_get_error() for more detail

Parameters

channel – pointer to the wolfSSH channel **buf** – buffer where wolfSSH_ChannelRead will place the data **bufSz** – size of the buffer

```
#include <wolfssh/ssh.h>
int wolfSSH_ChannelRead(WOLFSSH_CHANNEL* channel ,
byte* buf , word32 bufSz );
```

13.15.7 wolfSSH_ChannelSend()

Synopsis

Description

Sends data to the peer via the specified channel. Data is packaged into a channel data message. This will send as much data as possible via the peer socket. If there is more to be sent, calls to *wolfSSH_worker()* will continue sending more data for the channel to the peer.

Return Values

int – bytes sent **>0** – number of bytes sent upon success **0** – returns on socket failure cause by either a clean connection shutdown or a socket error, call wolfSSH_get_error() for more detail **WS_FATAL_ERROR** – there was some other error, call wolfSSH_get_error() for more detail

Parameters

channel – pointer to the wolfSSH channel **buf** – buffer wolfSSH_ChannelSend() will send **bufSz** – size of the buffer

```
#include <wolfssh/ssh.h>
int* wolfSSH_ChannelSend(WOLFSSH_CHANNEL* channel ,
const byte* buf , word32 bufSz );
```

13.15.8 wolfSSH_ChannelExit()

Synopsis

Description

Terminates a channel, sending the close message to the peer, marks the channel as closed. This does not free the channel and it remains on the channel list. After closure, data can not be sent on the channel, but data may still be available to be received. (At the moment, it sends EOF, close, and deletes the channel.)

Return Values

int – error code

Parameters

channel – wolfSSH session channel

```
#include <wolfssh/ssh.h>
int wolfSSH_ChannelExit(WOLFSSH_CHANNEL* channel );
```

13.15.9 wolfSSH_ChannelNext()

Synopsis

Description

Returns the next channel after *channel* in *ssh* 's channel list. If *channel* is NULL, the first channel from the channel list for *ssh* is returned.

Return Values

WOLFSSH_CHANNEL* – pointer to either the first channel, next channel, or NULL

Parameters

ssh – wolfSSH session **channel** – wolfSSH session channel

```
#include <wolfssh/ssh.h>
WOLFSSH_CHANNEL* wolfSSH_ChannelFwdNew(WOLFSSH* ssh ,
WOLFSSH_CHANNEL* channel );
```

13.16 Key Load Functions

13.16.1 wolfSSH_ReadKey_buffer()

Synopsis

```
#include <wolfssh/ssh.h>

int wolfSSH_ReadKey_buffer(const byte* in, word32 inSz,
        int format, byte** out, word32* outSz,
        const byte** outType, word32* outTypeSz,
        void* heap);
```

Description

Reads a key file from the buffer *in* of size *inSz* and tries to decode it as a *format* type key. The *format* can be **WOLFSSH_FORMAT_ASN1**, **WOLFSSH_FORMAT_PEM**, **WOLFSSH_FORMAT_SSH**, or **WOLFSSH_FORMAT_OPENSSH**. The key ready for use by `wolfSSH_UsePrivateKey_buffer()` is stored in the buffer pointed to by *out*, of size *outSz*. If *out* is NULL, *heap* is used to allocate a buffer for the key. The type string of the key is stored in *outType*, with its string length in *outTypeSz*.

Return Values

- **WS_SUCCESS** - read key is successful
- **WS_BAD_ARGUMENT** - parameter has a bad value
- **WS_MEMORY_E** - failure allocating memory
- **WS_BUFFER_E** - buffer not large enough for indicated size
- **WS_PARSE_E** - problem parsing the key file
- **WS_UNIMPLEMENTED_E** - key type not supported
- **WS_RSA_E** - something wrong with RSA (PKCS1) key
- **WS_ECC_E** - something wrong with ECC (X9.63) key
- **WS_KEY_AUTH_MAGIC_E** - OpenSSH key auth magic value bad
- **WS_KEY_FORMAT_E** - OpenSSH key format incorrect
- **WS_KEY_CHECK_VAL_E** - OpenSSH key check value corrupt

13.16.2 wolfSSH_ReadKey_file()

Synopsis

```
#include <wolfssh/ssh.h>

int wolfSSH_ReadKey_file(const char* name,
        byte** out, word32* outSz,
        const byte** outType, word32* outTypeSz,
        byte* isPrivate, void* heap);
```

Description

Reads the key from the file *name*. The format is guessed based on data in the file. The key buffer *out*, the key type *outType*, and their sizes are passed to `wolfSSH_ReadKey_buffer()`. The flag *isPrivate* is set as appropriate. Any memory allocations use the specified *heap*.

Return Values

- **WS_SUCCESS** - read key is successful
- **WS_BAD_ARGUMENT** - parameter has a bad value
- **WS_BAD_FILE_E** - problem reading the file
- **WS_MEMORY_E** - failure allocating memory
- **WS_BUFFER_E** - buffer not large enough for indicated size
- **WS_PARSE_E** - problem parsing the key file
- **WS_UNIMPLEMENTED_E** - key type not supported
- **WS_RSA_E** - something wrong with RSA (PKCS1) key
- **WS_ECC_E** - something wrong with ECC (X9.63) key
- **WS_KEY_AUTH_MAGIC_E** - OpenSSH key auth magic value bad
- **WS_KEY_FORMAT_E** - OpenSSH key format incorrect
- **WS_KEY_CHECK_VAL_E** - OpenSSH key check value corrupt

13.17 Key Exchange Algorithm Configuration

wolfSSH sets up a set of algorithm lists used during the Key Exchange (KEX) based on the availability of algorithms in the wolfCrypt library used.

Provided are some accessor functions to see which algorithms are available to use and to see the algorithm lists used in the KEX. The accessor functions come in sets of four: set or get from CTX object, and set or get from SSH object. All SSH objects made with a CTX inherit the CTX's algorithm lists, and they may be provided their own.

By default, any algorithms using SHA-1 are disabled but may be re-enabled using one of the following functions. If SHA-1 is disabled in wolfCrypt, then SHA-1 cannot be used.

13.17.1 wolfSSH Set Algo Lists

Synopsis

```
#include <wolfssh/ssh.h>

int wolfSSH_CTX_SetAlgoListKex(WOLFSSH_CTX* ctx, const char* list);
int wolfSSH_CTX_SetAlgoListKey(WOLFSSH_CTX* ctx, const char* list);
int wolfSSH_CTX_SetAlgoListCipher(WOLFSSH_CTX* ctx, const char* list);
int wolfSSH_CTX_SetAlgoListMac(WOLFSSH_CTX* ctx, const char* list);
int wolfSSH_CTX_SetAlgoListKeyAccepted(WOLFSSH_CTX* ctx, const char* list);

int wolfSSH_SetAlgoListKex(WOLFSSH* ssh, const char* list);
int wolfSSH_SetAlgoListKey(WOLFSSH* ssh, const char* list);
int wolfSSH_SetAlgoListCipher(WOLFSSH* ssh, const char* list);
int wolfSSH_SetAlgoListMac(WOLFSSH* ssh, const char* list);
```

```
int wolfSSH_SetAlgoListKeyAccepted(WOLFSSH* ssh, const char* list);
```

Description

These functions act as setters for the various algorithm lists set in the wolfSSH *ctx* or *ssh* objects. The strings are sent to the peer during the KEX Initialization and are used to compare against when the peer sends its KEX Initialization message. The KeyAccepted list is used for user authentication.

The CTX versions of the functions set the algorithm list for the specified WOLFSSH_CTX object, *ctx*. They have default values set at compile time. The specified value is used instead. Note, the library does not copy this string, it is owned by the application and it is up to the application to free it when the CTX is deallocated by the application. When creating an SSH object using a CTX, the SSH object inherits the CTX's strings. The SSH object algorithm lists may be overridden.

`Kex` specifies the key exchange algorithm list. `Key` specifies the server public key algorithm list. `Cipher` specifies the bulk encryption algorithm list. `Mac` specifies the message authentication code algorithm list. `KeyAccepted` specifies the public key algorithms allowed for user authentication.

Return Values

- **WS_SUCCESS** - successful
- **WS_SSH_CTX_NULL_E** - provided CTX was null
- **WS_SSH_NULL_E** - provide SSH was null

13.17.2 wolfSSH Get Algo List

Synopsis

```
#include <wolfssh/ssh.h>

const char* wolfSSH_CTX_GetAlgoListKex(WOLFSSH_CTX* ctx);
const char* wolfSSH_CTX_GetAlgoListKey(WOLFSSH_CTX* ctx);
const char* wolfSSH_CTX_GetAlgoListCipher(WOLFSSH_CTX* ctx);
const char* wolfSSH_CTX_GetAlgoListMac(WOLFSSH_CTX* ctx);
const char* wolfSSH_CTX_GetAlgoListKeyAccepted(WOLFSSH_CTX* ctx);

const char* wolfSSH_GetAlgoListKex(WOLFSSH* ssh);
const char* wolfSSH_GetAlgoListKey(WOLFSSH* ssh);
const char* wolfSSH_GetAlgoListCipher(WOLFSSH* ssh);
const char* wolfSSH_GetAlgoListMac(WOLFSSH* ssh);
const char* wolfSSH_GetAlgoListKeyAccepted(WOLFSSH* ssh);
```

Description

These functions act as getters for the various algorithm lists set in the wolfSSH *ctx* or *ssh* objects.

`Kex` specifies the key exchange algorithm list. `Key` specifies the server public key algorithm list. `Cipher` specifies the bulk encryption algorithm list. `Mac` specifies the message authentication code algorithm list. `KeyAccepted` specifies the public key algorithms allowed for user authentication.

Return Values

These functions return a pointer to either the default value set at compile time or the value set at run time with the setter functions. If the *ctx* or `ssh` parameters are NULL the functions return NULL.

13.17.3 wolfSSH_CheckAlgoName

Synopsis

```
#include <wolfssh/ssh.h>

int wolfSSH_CheckAlgoName(const char* name);
```

Description

Given a single algorithm *name* checks to see if it is valid.

Return Values

- **WS_SUCCESS** - *name* is a valid algorithm name
- **WS_INVALID_ALGO_ID** - *name* is an invalid algorithm name

13.17.4 wolfSSH Query Algorithms

Synopsis

```
#include <wolfssh/ssh.h>

const char* wolfSSH_QueryKex(word32* index);
const char* wolfSSH_QueryKey(word32* index);
const char* wolfSSH_QueryCipher(word32* index);
const char* wolfSSH_QueryMac(word32* index);
```

Description

Returns the name string for a valid algorithm of the particular type: Kex, Key, Cipher, or Mac. Note, Key types are also used for the user authentication accepted key types. The value passed as *index* must be initialized to 0, the passed in on each call to the function. At the end of the list, the *index* is invalid.

Return Values

Returns a constant string with the name of an algorithm. Null indicates the end of the list.

14 wolfSSL SFTP API Reference

14.1 Connection Functions

14.1.1 wolfSSH_SFTP_accept()

Synopsis:

Description:

Function to handle an incoming connection request from a client.

Return Values:

Returns WS_SFTP_COMPLETE on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_accept(WOLFSSH* ssh );

WOLFSSH* ssh;

//create new WOLFSSH structure
...

if (wolfSSH_SFTP_accept(ssh) != WS_SUCCESS) {
//handle error case
}
```

See Also:

wolfSSH_SFTP_free() wolfSSH_new() wolfSSH_SFTP_connect()

14.1.2 wolfSSH_SFTP_connect()

Synopsis:

Description:

Function for initiating a connection to a SFTP server.

Return Values:

WS_SFTP_COMPLETE: on success.

Parameters:

ssh - pointer to WOLFSSH structure to be used for connection

Example:

See Also:

wolfSSH_SFTP_accept() wolfSSH_new() wolfSSH_free()

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_connect(WOLFSSH* ssh );

WOLFSSH* ssh;
```

```
//after creating a new WOLFSSH structrue

wolfSSH_SFTP_connect(ssh);
```

14.1.3 wolfSSH_SFTP_negotiate()

Synopsis:

Description:

Function to handle either an incoming connection from client or to send out a connection request to a server. It is dependent on which side of the connection the created WOLFSSH structure is set to for which action is performed.

Return Values:

Returns WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection

Example:

See Also:

wolfSSH_SFTP_free()

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_negotiate(WOLFSSH* ssh)

WOLFSSH* ssh;

//create new WOLFSSH structure with side of connection
set
....

if (wolfSSH_SFTP_negotiate(ssh) != WS_SUCCESS) {
//handle error case
}
```

wolfSSH_new() wolfSSH_SFTP_connect() wolfSSH_SFTP_accept()

14.2 Protocol Level Functions

14.2.1 wolfSSH_SFTP_RealPath()

Synopsis:

Description:

Function to send REALPATH packet to peer. It gets the name of the file returned from peer.

Return Values:

Returns a pointer to a WS_SFTPNAME structure on success and NULL on error.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **dir** - directory / file name to get real path of

Example:

```
#include <wolfssh/wolfsftp.h>
WS_SFTPNAME* wolfSSH_SFTP_RealPath(WOLFSSH* ssh , char*
dir );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh ;

//set up ssh and do sftp connections
...

if (wolfSSH_SFTP_read( ssh ) != WS_SUCCESS) {
//handle error case
}
```

14.2.2 wolfSSH_SFTP_Close()

Synopsis:

Description:

Function to to send a close packet to the peer.

Return Values:

WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **handle** - handle to try and close **handleSz** - size of handle buffer

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_Close(WOLFSSH* ssh , byte* handle , word32
handleSz );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh;
byte handle[HANDLE_SIZE];
word32 handleSz = HANDLE_SIZE;

//set up ssh and do sftp connections
...

if (wolfSSH_SFTP_Close(ssh, handle, handleSz) !=
WS_SUCCESS) {
//handle error case
}
```

14.2.3 wolfSSH_SFTP_Open()

Synopsis:

Description:

Function to to send an open packet to the peer. This sets handleSz with the size of resulting buffer and gets the resulting handle from the peer and places it in the buffer handle.

Available reasons for open: WOLFSSH_FXF_READ WOLFSSH_FXF_WRITE WOLFSSH_FXF_APPEND WOLFSSH_FXF_CREAT WOLFSSH_FXF_TRUNC WOLFSSH_FXF_EXCL

Return Values:

WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **dir** - name of file to open **reason** - reason for opening the file **atr** - initial attributes for file **handle** - resulting handle from open **handleSz** - gets set to the size of resulting handle

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_Open(WOLFSSH* ssh , char* dir , word32
reason ,
WS_SFTP_FILEATRB* atr , byte* handle , word32* handleSz ) ;
```

Example:

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh ;
char name[NAME_SIZE];
byte handle[HANDLE_SIZE];
word32 handleSz = HANDLE_SIZE;
WS_SFTP_FILEATRB atr;

//set up ssh and do sftp connections
...

if (wolfSSH_SFTP_Open( ssh , name , WOLFSSH_FXF_WRITE |
WOLFSSH_FXF_APPEND | WOLFSSH_FXF_CREAT , & atr , handle ,
& handleSz )
!= WS_SUCCESS) {
//handle error case
}
```

14.2.4 wolfSSH_SFTP_SendReadPacket()

Synopsis:

Description:

Function to to send a read packet to the peer. The buffer handle should contain the result of a previous call to wolfSSH_SFTP_Open. The resulting bytes from a read are placed into the “out” buffer.

Return Values:

Returns the number of bytes read on success. A negative value is returned on failure.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **handle** - handle to try and read from **handleSz** - size of handle buffer **ofst** - offset to start reading from **out** - buffer to hold result from read **outSz** - size of out buffer

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_SendReadPacket(WOLFSSH* ssh , byte*
handle , word32
handleSz , word64 ofst , byte* out , word32 outSz );
```

See Also:

wolfSSH_SFTP_SendWritePacket() wolfSSH_SFTP_Open()

```
WOLFSSH* ssh;
byte handle[HANDLE_SIZE];
word32 handleSz = HANDLE_SIZE;
byte out[OUT_SIZE];
word32 outSz = OUT_SIZE;
word32 ofst = 0;
int ret;

//set up ssh and do sftp connections
...
//get handle with wolfSSH_SFTP_Open()

if ((ret = wolfSSH_SFTP_SendReadPacket(ssh, handle,
handleSz, ofst,
out, outSz)) < 0) {
//handle error case
}
//ret holds the number of bytes placed into out buffer
```

14.2.5 wolfSSH_SFTP_SendWritePacket()

Synopsis:

Description:

Function to send a write packet to the peer. The buffer handle should contain the result of a previous call to wolfSSH_SFTP_Open().

Return Values:

Returns the number of bytes written on success. A negative value is returned on failure.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **handle** - handle to try and read from **handleSz** - size of handle buffer **ofst** - offset to start reading from **out** - buffer to send to peer for writing **outSz** - size of out buffer

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_SendWritePacket(WOLFSSH* ssh , byte*
handle , word32
handleSz , word64 ofst , byte* out , word32 outSz );
```

See Also:

wolfSSH_SFTP_SendReadPacket() wolfSSH_SFTP_Open()

```
WOLFSSH* ssh;
byte handle[HANDLE_SIZE];
word32 handleSz = HANDLE_SIZE;
byte out[OUT_SIZE];
word32 outSz = OUT_SIZE;
word32 ofst = 0;
int ret;

//set up ssh and do sftp connections
...
//get handle with wolfSSH_SFTP_Open()

if ((ret = wolfSSH_SFTP_SendWritePacket(ssh, handle,
handleSz, ofst,
out,outSz)) < 0) {
//handle error case
}
//ret holds the number of bytes written
```

14.2.6 wolfSSH_SFTP_STAT()

Synopsis:

Description:

Function to send a STAT packet to the peer. This will get the attributes of file or directory. If the file or attribute does not exist the peer will return resulting in this function returning an error value.

Return Values:

WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **dir** - NULL terminated name of file or directory to get attributes of **atr** - resulting attributes are set into this structure

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_STAT(WOLFSSH* ssh , char* dir ,
WS_SFTP_FILEATRB* atr);
```

See Also:

wolfSSH_SFTP_LSTAT() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh;
byte name[NAME_SIZE];
int ret;
WS_SFTP_FILEATRB atr;

//set up ssh and do sftp connections
...
```

```
if ((ret = wolfSSH_SFTP_STAT(ssh, name, &atr)) < 0) {
//handle error case
}
```

14.2.7 wolfSSH_SFTP_LSTAT()

Synopsis:

Description:

Function to send a LSTAT packet to the peer. This will get the attributes of file or directory. It follows symbolic links where a STAT packet will not follow symbolic links. If the file or attribute does not exist the peer will return resulting in this function returning an error value.

Return Values:

WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **dir** - NULL terminated name of file or directory to get attributes of **atr** - resulting attributes are set into this structure

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_LSTAT(WOLFSSH* ssh , char* dir ,
WS_SFTP_FILEATRB* atr );
```

See Also:

wolfSSH_SFTP_STAT() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh;
byte name[NAME_SIZE];
int ret;
WS_SFTP_FILEATRB atr;

//set up ssh and do sftp connections
...

if ((ret = wolfSSH_SFTP_LSTAT(ssh, name, &atr)) < 0) {
//handle error case
}
```

14.2.8 wolfSSH_SFTPNAME_free()

Synopsis:

Description:

Function to free a single WS_SFTPNAME node. Note that if this node is in the middle of a list of nodes then the list will be broken.

Return Values:

None

Parameters:

name - structure to be free'd

Example:

See Also:

```
#include <wolfssh/wolfsftp.h>
```

14.2.9 void wolfSSH_SFTPNAME_free(WS_SFTPNMAE* name);

```
WOLFSSH* ssh;
WS_SFTPNAME* name;

//set up ssh and do sftp connections
...
name = wolfSSH_SFTP_RealPath(ssh, path);
if (name != NULL) {
wolfSSH_SFTPNAME_free(name);
}
```

wolfSSH_SFTPNAME_list_free

wolfSSH_SFTPNAME_list_free()

Synopsis:

Description:

Function to free a all WS_SFTPNAME nodes in a list.

Return Values:

None

Parameters:

name - head of list to be free'd

Example:

```
#include <wolfssh/wolfsftp.h>
void wolfSSH_SFTPNAME_list_free(WS_SFTPNMAE* name );
```

See Also:

wolfSSH_SFTPNAME_free()

```
WOLFSSH* ssh;
WS_SFTPNAME* name;

//set up ssh and do sftp connections
...

name = wolfSSH_SFTP_LS(ssh, path);
if (name != NULL) {
wolfSSH_SFTPNAME_list_free(name);
}
```

14.3 Reget / Reput Functions

14.3.1 wolfSSH_SFTP_SaveOfst()

Synopsis:

Description:

Function to save an offset for an interrupted get or put command. The offset can be recovered by calling wolfSSH_SFTP_GetOfst

Return Values:

Returns WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure for connection **from** - NULL terminated string of source path **to** - NULL terminated string with destination path **ofst** - offset into file to be saved

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_SaveOfst(WOLFSSH* ssh , char* from , char*
to ,
word64 ofst );
```

See Also:

wolfSSH_SFTP_GetOfst() wolfSSH_SFTP_Interrupt()

```
WOLFSSH* ssh;
char from[NAME_SZ];
char to[NAME_SZ];
word64 ofst;

//set up ssh and do sftp connections
...

if (wolfSSH_SFTP_SaveOfst(ssh, from, to, ofst) !=
WS_SUCCESS) {
//handle error case
}
```

14.3.2 wolfSSH_SFTP_GetOfst()

Synopsis:

Description:

Function to retrieve an offset for an interrupted get or put command.

Return Values:

Returns offset value on success. If not stored offset is found then 0 is returned.

Parameters:

ssh - pointer to WOLFSSH structure for connection **from** - NULL terminated string of source path **to** - NULL terminated string with destination path

Example:

```
#include <wolfssh/wolfsftp.h>
word64 wolfSSH_SFTP_GetOfst(WOLFSSH* ssh, char* from,
char* to);
```

```
WOLFSSH* ssh;
char from[NAME_SZ];
char to[NAME_SZ];
word64 ofst;

//set up ssh and do sftp connections
...

ofst = wolfSSH_SFTP_GetOfst(ssh, from, to);
//start reading/writing from ofst
```

See Also:

wolfSSH_SFTP_SaveOfst() wolfSSH_SFTP_Interrup()

14.3.3 wolfSSH_SFTP_ClearOfst()

Synopsis:

Description:

Function to clear all stored offset values.

Return Values:

WS_SUCCESS on success

Parameters:

ssh - pointer to WOLFSSH structure

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_ClearOfst(WOLFSSH* ssh);
```

See Also:

wolfSSH_SFTP_SaveOfst() wolfSSH_SFTP_GetOfst()

14.3.4 wolfSSH_SFTP_Interrupt()

Synopsis:

Description:

Function to set interrupt flag and stop a get/put command.

Return Values:

None

Parameters:

ssh - pointer to WOLFSSH structure

Example:

```
WOLFSSH* ssh;

//set up ssh and do sftp connections
...
```

```
if (wolfSSH_SFTP_ClearOfst(ssh) != WS_SUCCESS) {
//handle error
}

#include <wolfssh/wolfsftp.h>
void wolfSSH_SFTP_Interrupt(WOLFSSH* ssh);
```

See Also:

wolfSSH_SFTP_SaveOfst() wolfSSH_SFTP_GetOfst()

```
WOLFSSH* ssh;
char from[NAME_SZ];
char to[NAME_SZ];
word64 ofst;

//set up ssh and do sftp connections
...

wolfSSH_SFTP_Interrupt(ssh);
wolfSSH_SFTP_SaveOfst(ssh, from, to, ofst);
```

14.4 Command Functions

14.4.1 wolfSSH_SFTP_Remove()

Synopsis:

Description:

Function for sending a “remove” packet across the channel. The file name passed in as “f” is sent to the peer for removal.

Return Values:

WS_SUCCESS : returns WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **f** - file name to be removed

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_Remove(WOLFSSH* ssh , char* f );

WOLFSSH* ssh;
int ret;
char* name[NAME_SZ];

//set up ssh and do sftp connections
...

ret = wolfSSH_SFTP_Remove(ssh, name);
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

14.4.2 wolfSSH_SFTP_MKDIR()

Synopsis:

Description:

Function for sending a “mkdir” packet across the channel. The directory name passed in as “dir” is sent to the peer for creation. Currently the attributes passed in are not used and default attributes is set instead.

Return Values:

WS_SUCCESS : returns WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection dir - NULL terminated directory to be created atr - attributes to be used with directory creation

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_MKDIR(WOLFSSH* ssh , char* dir ,
WS_SFTP_FILEATRB*
atr );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

14.4.3 wolfSSH_SFTP_RMDIR()

Synopsis:

Description:

Function for sending a “rmdir” packet across the channel. The directory name passed in as “dir” is sent to the peer for deletion.

Return Values:

WS_SUCCESS : returns WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **dir** - NULL terminated directory to be remove

```
WOLFSSH* ssh;
int ret;
char* dir[DIR_SZ];

//set up ssh and do sftp connections
...

ret = wolfSSH_SFTP_MKDIR(ssh, dir, DIR_SZ);

#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_RMDIR(WOLFSSH* ssh , char* dir );
```

Example:

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh;
int ret;
char* dir[DIR_SZ];

//set up ssh and do sftp connections
...

ret = wolfSSH_SFTP_RMDIR(ssh, dir);
```

14.4.4 wolfSSH_SFTP_Rename()

Synopsis:

Description:

Function for sending a “rename” packet across the channel. This tries to have a peer file renamed from “old” to “nw”.

Return Values:

WS_SUCCESS : returns WS_SUCCESS on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **old** - Old file name **nw** - New file name

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_Rename(WOLFSSH* ssh , const char* old ,
const char*
nw );

WOLFSSH* ssh;
int ret;
char* old[NAME_SZ];
char* nw[NAME_SZ]; //new file name

//set up ssh and do sftp connections
...

ret = wolfSSH_SFTP_Rename(ssh, old, nw);
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

14.4.5 wolfSSH_SFTP_LS()

Synopsis:

Description:

Function for performing LS operation which gets a list of all files and directories in the current working directory. This is a high level function that performs REALPATH, OPENDIR, READDIR, and CLOSE operations.

Return Values:

On Success, returns a pointer to a list of WS_SFTPNAME structures. NULL on failure.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **dir** - directory to list

Example:

```
#include <wolfssh/wolfsftp.h>
WS_SFTPNAME* wolfSSH_SFTP_LS(WOLFSSH* ssh , char* dir );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect() wolfSSH_SFTPNAME_list_free()

```
WOLFSSH* ssh;
int ret;
char* dir[DIR_SZ];
WS_SFTPNAME* name;
WS_SFTPNAME* tmp;

//set up ssh and do sftp connections
...

name = wolfSSH_SFTP_LS(ssh, dir);
tmp = name;
while (tmp != NULL) {
printf("%s\n", tmp->fName);
tmp = tmp->next;
}
wolfSSH_SFTPNAME_list_free(name);
```

14.4.6 wolfSSH_SFTP_Get()

Synopsis:

Description:

Function for performing get operation which gets a file from the peer and places it in a local directory. This is a high level function that performs LSTAT, OPEN, READ, and CLOSE operations. To interrupt the operation call the function wolfSSH_SFTP_Interrupt. (See the API documentation of this function for more information on what it does)

Return Values:

WS_SUCCESS : on success. All other return values should be considered error cases.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **from** - file name to get **to** - file name to place result at **resume** - flag to try resume of operation. 1 for yes 0 for no **statusCb** - callback function to get status

Example:

```
#include <wolfssh/wolfsftp.h>

int wolfSSH_SFTP_Get(WOLFSSH* ssh , char* from , char* to ,
byte resume ,
WS_STATUS_CB* statusCb );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
static void myStatusCb(WOLFSSH* sshIn, long bytes, char*
name)
{
char buf[80];
WSNPRINTF(buf, sizeof(buf), "Processed %8ld\t bytes
\r", bytes);
WFPUTS(buf, fout);
(void)name;
(void)sshIn;
}
...
WOLFSSH* ssh;
char* from[NAME_SZ];
char* to[NAME_SZ];

//set up ssh and do sftp connections
...

if (wolfSSH_SFTP_Get( ssh , from , to , 0 , & myStatusCb ) !=
WS_SUCCESS) {
//handle error case
}
```

14.4.7 wolfSSH_SFTP_Put()

Synopsis:

Description:

Function for performing put operation which pushes a file local file to a peers directory. This is a high level function that performs OPEN, WRITE, and CLOSE operations. To interrupt the operation call the function wolfSSH_SFTP_Interrupt. (See the API documentation of this function for more information on what it does)

Return Values:

WS_SUCCESS on success. All other return values should be considered error cases.

Parameters:

ssh - pointer to WOLFSSH structure used for connection **from** - file name to push **to** - file name to place result at **resume** - flag to try resume of operation. 1 for yes 0 for no **statusCb** - callback function to get status

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_Put(WOLFSSH* ssh , char* from , char* to ,
byte resume , WS_STATUS_CB* statusCb );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
static void myStatusCb(WOLFSSH* sshIn, long bytes, char*
name)
{
char buf[80];
WSNPRINTF(buf, sizeof(buf), "Processed %8ld\t bytes
```

```
\r", bytes);
WFPUTS(buf, fout);
(void)name;
(void)sshIn;
}
...

WOLFSSH* ssh;
char* from[NAME_SZ];
char* to[NAME_SZ];

//set up ssh and do sftp connections
...

if (wolfSSH_SFTP_Put(ssh, from, to, 0, &myStatusCb) !=
WS_SUCCESS) {
//handle error case
}
```

14.5 SFTP Server Functions

14.5.1 wolfSSH_SFTP_read()

Synopsis:

Description:

Main SFTP server function that handles incoming packets. This function tries to read from the I/O buffer and calls internal functions to depending on the SFTP packet type received.

Return Values:

WS_SUCCESS: on success.

Parameters:

ssh - pointer to WOLFSSH structure used for connection

Example:

```
#include <wolfssh/wolfsftp.h>
int wolfSSH_SFTP_read(WOLFSSH* ssh );
```

See Also:

wolfSSH_SFTP_accept() wolfSSH_SFTP_connect()

```
WOLFSSH* ssh;

//set up ssh and do sftp connections
...
if (wolfSSH_SFTP_read(ssh) != WS_SUCCESS) {
//handle error case
}
```