wolfCLU Documentation

2026-07-07

Contents

1 wolfCLU Manual

wolfSSL's Command Line Utility (version 0.1.8) April 4, 2025

1.1 Intro

wolfCLU was created to handle some common cryptographic operations to make it easier/quicker than writing an application from scratch. An example of some of the operations handled are certificate parsing and key generation.

1.2 Building wolfCLU

1.2.1 Building on *NIX

To build wolfCLU, start by building wolfSSL with the –enable-wolfclu flag. An example of this would be:

```
cd wolfssl
./configure --enable-wolfclu
make
sudo make install
```

Note that if parsing PKCS12 files with RC2 or if using CRL the flags –enable-rc2 and –enable-crl would also need to be used when building wolfSSL.

Then build wolfCLU linking agianst the wolfSSL library created.

```
cd wolfclu
./configure
make
sudo make install

or

cd wolfclu
./configure --with-wolfssl=/path/to/wolfssl/install
make
sudo make install
```

Run `make check` to run unit tests.

1.3 Building on Windows

wolfCLU can also be built with the appropriate Visual Studio solution, wolfclu.sln. The solution provides both Debug and Release builds of Dynamic 32- or 64-bit libraries. The file `user_settings.h` should be used in the wolfSSL build to configure it.

The file `wolfclu\ide\winvs\user_settings.h` contains the settings used to configure wolfSSL with the appropriate settings. This file must be copied from the directory `wolfclu\ide\winvs` to `wolfssl\IDE\WIN`. You can then build wolfSSL with support for wolfCLU.

Before building wolfCLU, make sure you have the same architecture (Win32 or x64) selected as used in wolfSSL.

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\
    wolfclu\
    wolfssl\
```

Building a wolfCLU release configuration will generate `wolfssl.exe` in the `Release\Win32` or `Release\x64` directory.

1.3.0.1 Running Unit Tests To run the shell-script unit tests, you will need either the `sh` or `bash` commands, both of which come with a Git installation on Windows (although you may have to add them to the PATH).

1. Copy the `wolfssl.exe` to the root directory of wolfclu.
2. Modify the `run` function (as well as `run_fail`, if present) of the desired unit test to run `./wolfssl.exe $1` instead of `./wolfssl $1`.
3. In your terminal, run `sh <desired_unit_test>` from the root directory. For instance, to run the hash unit tests, run `sh tests\hash\hash-test.sh`.

1.4 List Of Commands

- base64
- bench
- ca
- crl
- dgst
- dhparam
- dsaparam
- ecc
- ecparam
- ed25519
- enc / encrypt / decrypt
- genkey
- hash
- md5
- pkcs7
- pkcs8
- pkcs12
- pkey
- rand
- req
- rsa
- s_client
- s_server
- sha256
- sha384
- sha512
- verify
- version
- x509

1.4.1 BENCH Command

Used for benchmarking algorithms. To benchmark all algorithms run `wolfssl bench -a`.

Available algorithms (depends on configure settings):

- aes-cbc
- aes-ctr
- 3des
- camellia
- md5
- sha
- sha256
- sha384
- sha512
- blake2b

Arguments:

- [algorithm] algorithm to benchmark
- [-time] time in seconds to run (1-10)
- [-a] or [-all] test all available algorithms
- [-in] input file to read from
- [-out] output file to write to

Example:

```
wolfssl bench aes-cbc -time 10

wolfssl bench -a -time 5
```

1.4.2 CA Command

Used for signing Certificates. Can handle some basic config file parsing.

See Configuration File for details on config file directives.

Arguments:

- [-in] input CSR file
- [-out] file to write to
- [-keyfile] file to read private key from
- [-cert] file to read CA from
- [-extensions] section in config file to parse extensions from
- [-md] type of hash to use i.e sha, sha256, ...
- [-inform] PEM/DER type of CSR input
- [-config] file to parse for configuration
- [-days] number of days should be valid for
- [-selfsign] sign with key associated with input cert

Example:

```
wolfssl ca -config ca.conf -in test.csr -out test.pem -md sha256 -selfsign -
   keyfile ./key
```

1.4.3 Typical Workflow: Sign CSR with CA

Step 1: Create CA key and certificate (see GENKEY Command and REQ Command)

```
wolfssl genkey rsa -size 2048 -out ca -outform pem -output KEYPAIR
wolfssl req -new -x509 -key ca.priv -out ca.pem -days 3650 -subj "/C=US/ST=
   Washington/L=Seattle/O=wolfSSL/CN=wolfSSL CA"
```

Step 2: Create server key and CSR

```
wolfssl genkey rsa -size 2048 -out server -outform pem -output KEYPAIR
wolfssl req -new -key server.priv -out server.csr -subj "/C=US/ST=Washington/L
   =Seattle/O=wolfSSL/CN=example.com"
```

Step 3: Sign server CSR with CA

```
wolfssl ca -in server.csr -out server-signed.pem -keyfile ca.priv -cert ca.pem
    -days 365 -md sha256
```

Step 4: Verify the signed certificate (see VERIFY Command)

```
wolfssl verify -CAfile ca.pem server-signed.pem
```

1.4.4 Using Configuration File with CA

The `-config` option allows specifying certificate extensions to be added when signing.

Example CA config file (`ca.conf`):

```
[ca]
x509_extensions = v3_ca

[v3_ca]
basicConstraints = critical, CA:FALSE
keyUsage = digitalSignature, keyEncipherment
subjectKeyIdentifier = hash
subjectAltName = @alt_names

[alt_names]
DNS.1 = example.com
DNS.2 = www.example.com
```

Sign CSR with extensions from config:

```
wolfssl ca -config ca.conf -extensions v3_ca -in server.csr -out server-signed
   .pem -keyfile ca.priv -cert ca.pem -days 365 -md sha256
```

1.4.5 CRL Command

Used to verify a CRL file given a CA, or to convert a CRL from one format [DER | PEM] to the other. The command will print out the CRL to stdout if -out is not specified and -noout is not used. Prints out "OK" on successful verification.

Arguments:

- [-CAfile] CA file name
- [-inform] pem or der in format
- [-in] the file to read from
- [-outform] pem or der out format
- [-out] output file to write to
- [-noout] do not print output if set
- [-text] output human readable text of CRL

Example:

```
wolfssl crl -CAfile ./certs/ca-cert.pem -in ./certs/crl.der -inform DER -noout

wolfssl crl -in ./certs/crl.pem -text
```


1.4.6 DSAPARAM Command

Used for creating DSA params and keys. Make sure wolfSSL is compiled with `--enable-dsa`.

Available arguments are:

- [-genkey] create new DSA key
- [-in] file to read params from to create a key
- [-out] file to output to (default stdout)
- [-noout] do not print out the params

Example:

```
wolfssl dsaparam -out dsa.params 1024
```

```
wolfssl dsaparam -in dsa.params -genkey
```

1.4.7 DGST Command

Used for creating and verifying digital signatures. The last argument is the data that was signed.

Hash algorithms supported:

- [-md5]
- [-sha]
- [-sha224]
- [-sha256]
- [-sha384]
- [-sha512]

Arguments:

- [-signature] file containing the signature
- [-inform] pem or der in format
- [-verify] key used to verify the signature
- [-sign] private key used to create the signature
- [-out] output file for signature

Sign

Example:

```
wolfssl dgst -sha256 -sign keyPrivate.pem -out test.sig testfile
```

Verify

Example:

```
wolfssl dgst -sha256 -signature test.sig -verify keyPublic.pem testfile
```

1.4.8 DHPARAM Command

Used for creating Diffie Hellman params and keys.

Available arguments are:

- [-genkey] create new DH key
- [-in] file to read params from to create a key
- [-out] file to output to (default stdout)
- [-check] check if generated parameters are valid
- [-noout] do not print out the params

Example:

```
wolfssl dhparam -check -out dh.params 1024
```

1.4.9 ECC Command

Sign and verify data using ECC (Elliptic Curve Cryptography) keys.

Arguments:

- [-sign] Sign operation
- [-verify] Verify operation
- [-inkey] Input key file
- [-in] Input file to sign or verify
- [-out] Output file for signature (sign operation)
- [-sigfile] Signature file (verify operation)
- [-pubin] Input key is a public key (for verify operation)

Usage Examples:

Sign a file with ECC private key:

```
wolfssl ecc -sign -inkey ecc-priv.pem -in file.txt -out file.sig
```

Verify a signature with ECC public key:

```
wolfssl ecc -verify -inkey ecc-pub.pem -sigfile file.sig -in file.txt -pubin
```

Verify a signature with ECC private key:

```
wolfssl ecc -verify -inkey ecc-priv.pem -sigfile file.sig -in file.txt
```

1.4.10 ECPARAM Command

Used for creating ECC keys.

Arguments:

- [-genkey] create new key
- [-out] output file
- [-name] curve name

Available curve names:

- PRIME239V1
- PRIME239V2
- PRIME239V3
- SECP256R1
- SECP224R1
- SECP384R1
- SECP521R1
- SECP256K1
- BRAINPOOLP224R1
- BRAINPOOLP256R1
- BRAINPOOLP320R1
- BRAINPOOLP384R1
- BRAINPOOLP512R1

Example:

```
wolfssl ecparam -genkey -out new.key -name secp384r1
```

```
wolfssl ecparam -genkey -out brainpool.key -name BRAINPOOLP256R1
```

1.4.11 ED25519 Command

Sign and verify data using Ed25519 keys.

Arguments:

- [-sign] Sign operation
- [-verify] Verify operation
- [-inkey] Input key file
- [-in] Input file to sign or verify
- [-out] Output file for signature (sign operation)
- [-sigfile] Signature file (verify operation)
- [-pubin] Input key is a public key (for verify operation)

Usage Examples:

Sign a file with Ed25519 private key:

```
wolfssl ed25519 -sign -inkey ed25519-priv.pem -in file.txt -out file.sig
```

Verify a signature with Ed25519 public key:

```
wolfssl ed25519 -verify -inkey ed25519-pub.pem -sigfile file.sig -in file.txt
    -pubin
```

Verify a signature with Ed25519 private key:

```
wolfssl ed25519 -verify -inkey ed25519-priv.pem -sigfile file.sig -in file.txt
```

1.4.12 ENC Command

Used for encrypting an input. Setting -d enables decryption.

Available encryption and decryption algorithms (depends on configure settings):

- aes-cbc-128
- aes-cbc-192
- aes-cbc-256
- aes-ctr-128
- aes-ctr-192
- aes-ctr-256
- 3des-cbc-56
- 3des-cbc-112
- 3des-cbc-168
- camellia-cbc-128
- camellia-cbc-192
- camellia-cbc-256

Arguments:

- [-in] input file to read from
- [-out] file to write to (default stdout)
- [-pwd] password input
- [-k] another option for password input
- [-pass] option for password source, e.g., pass:password

- [-key] hex key input
- [-iv] hex iv input
- [-inkey] input file for key
- [-pbkdf2] use kdf version 2
- [-md] specify hash algo to use, e.g., md5, sha256
- [-d] decrypt the input file
- [-p] display debug information (key / iv ...)
- [-base64] handle decoding a base64 input
- [-nosalt] do not use a salt input to kdf

Example:

```
wolfssl enc -aes-128-cbc -k Thi$i$myPa$$w0rd -in somefile.txt

wolfssl enc aes-cbc-256 -pwd Thi$i$myPa$$w0rd -in somefile.txt -out
   encryptedfile.txt

wolfssl enc aes-cbc-256 -d -pwd Thi$i$myPa$$w0rd -in encryptedfile.txt -out
   decryptedfile.txt
```

1.4.13 GENKEY Command

Used to generate RSA, ECC and ED25519 keys. If using "-output KEYPAIR" a private key is created having .priv appended to -out argument and a public key is created with .pub appended to the -out argument. If generating ED25519 keys compile wolfSSL with –enable-ed25519.

Available key types (depends on configure settings):

- rsa
- ecc
- ed25519

Arguments:

- [rsa | ecc | ed25519] key type to generate
- [-size] size of key to generate (bits)
- [-out] file to write to
- [-outform] output form, either DER or PEM (defaults to DER)
- [-output] key to generate, either PUB, PRIV or KEYPAIR (defaults to KEYPAIR)

Example:

```
wolfssl genkey rsa -size 2048 -out mykey -outform pem -output KEYPAIR

wolfssl genkey ecc -out ecckey -outform pem -output KEYPAIR

wolfssl genkey ed25519 -out ed25519key -outform pem -output KEYPAIR
```

The above command would output the files: mykey.priv and mykey.pub. Changing the -output option to just PRIV would output only the private key.

1.4.14 Typical Workflow: RSA Key to Certificate

Step 1: Generate RSA key pair for server

```
wolfssl genkey rsa -size 2048 -out server -outform pem -output KEYPAIR
```

This creates `server.priv` (private key) and `server.pub` (public key).

Step 2: Generate RSA key pair for CA

```
wolfssl genkey rsa -size 2048 -out ca -outform pem -output KEYPAIR
```

See the REQ Command for creating CSR and certificates, and the CA Command for signing certificates.

1.4.15 HASH Command

Used to create a hash of input data.

Available algorithms (depends on configure settings):

- md5
- sha
- sha256
- sha384
- sha512
- blake2b
- base64enc
- base64dec

Arguments:

- [algorithm] hash algorithm to use
- [-in] file to hash

Example:

```
wolfssl hash sha256 -in somefile.txt

wolfssl hash blake2b -in somefile.txt
```

1.4.16 MD5 Command

Used to create an MD5 hash of input data. The last argument is the file to be hashed, if a file argument is not provided then stdin is used. Note that when using stdin the input must be provided upon calling wolfssl. A correct usage and incorrect usage of stdin are shown below:

Incorrect :

```
wolfssl md5
> hi
>
```

Correct :

```
echo "hi" | wolfssl md5
```

The reason the incorrect version is wrong is because no output will be received. This input method is used because it mimics the usage of OpenSSL's CLI, thus it encourages portability when switching from OpenSSL to WolfSSL.

Examples :

```
wolfssl md5 configure.ac

978425cba5277d73db2a76d72b523d48
```

```
echo "hi" | wolfssl md5

764efa883dda1e11db47671c4a3bbd9e
```

1.4.17 PKCS12 Command

Currently only PKCS12 parsing is supported and PKCS12 generation is not yet supported. By default the –enable-wolfclu option used when building wolfSSL has PKCS12 support enabled but it does not enable RC2. If parsing PKCS12 bundles that have been encrypted using RC2 then –enable-rc2 should also be used when compiling wolfSSL.

- [-in] file input for pkcs12 bundle
- [-out] file to output results to (default stdout)
- [-nodes] no DES encryption
- [-nocerts] no certificate output
- [-nokeys] no key output
- [-passin] source to get password from
- [-passout] source to output password to

Example:

```
wolfssl pkcs12 -nodes -passin pass:"wolfSSL test" -in ./certs/test-servercert.
    p12
```

1.4.18 PKCS7 Command

Processes PKCS#7 data, allowing operations such as extracting certificates from PKCS#7 files. PKCS#7 is a standard for cryptographically signed and/or encrypted data.

- [-in] input file containing PKCS#7 data (required)
- [-out] output file to write results to (default stdout)
- [-inform] input format (PEM or DER, default PEM)
- [-outform] output format (PEM or DER, default PEM)
- [-print_certs] extract and output certificates from the PKCS#7 file

Example for extracting certificates from a PKCS#7 file:

```
wolfssl pkcs7 -in pkcs7.pem -print_certs
```

Example for converting PKCS#7 data from PEM to DER format:

```
wolfssl pkcs7 -in pkcs7.pem -outform DER -out pkcs7.der
```

1.4.19 PKCS8 Command

Processes PKCS#8 private key files. Allows conversion between different formats (PEM/DER) and decryption of private keys. Encrypting PKCS#8 keys is not yet supported.

- [-in] input file containing the private key (required)
- [-out] output file to write the processed key to (default stdout)
- [-inform] input format (PEM or DER, default PEM)
- [-outform] output format (PEM or DER, default PEM)
- [-passin] password source for encrypted input key
- [-traditional] output key in traditional (non-PKCS#8) format
- [-topk8] convert input to PKCS#8 format
- [-nocrypt] don’t encrypt the output key (no password)

Example for converting an encrypted PEM key to DER format:

```
wolfssl pkcs8 -in server-keyEnc.pem -passin pass:mypassword -outform DER -out
    key.der
```

Example for converting a key to traditional format:

```
wolfssl pkcs8 -in server-key.pem -traditional -out traditional-key.pem
```

1.4.20 PKEY Command

Used for dealing with generic key operations. Prints the key read in to stdout.

Arguments:

- [-in] file input for key to read
- [-out] file to output to (default stdout)
- [-inform] pem or der input format (defaults to pem)
- [-outform] pem or der output format
- [-pubout] output the public key
- [-pubin] expect to read public key in

Example:

```
wolfssl pkey -in ./certs/server-key.pem -inform pem -pubout

wolfssl pkey -in ./certs/server-key.pem -out pubkey.pem -pubout
```

1.4.21 RAND Command

Generates random bytes in raw or base64 form. By default it outputs the result to stdout but can be redirected with the '-out' argument. The last argument passed in is the number of random bytes to generate.

- [-base64] base64 encode the resulting random bytes
- [-out] output file to write results to

Example:

```
wolfssl rand -base64 10
```

1.4.22 REQ Command

Used for creating a certificate request or a self-signed certificate. Can handle some basic parsing of a .conf file for certificate setup. If no configuration file is used then stdin is prompted for certificate information.

See Configuration File for details on the config file directives.

Arguments:

- [-in] input file to read from
- [-out] file to write to (default stdout)
- [-key] public key to put into certificate request
- [-inform] der or pem format for '-in' (defaults to pem)
- [-outform] der or pem format for '-out' (defaults to pem)
- [-config] file to parse for certificate configuration
- [-days] number of days should be valid for

- [-x509] generate self signed certificate
- [-extensions] overwrite the section to get extensions from
- [-nodes] no DES encryption on private key output
- [-newkey] generate the private key to use with req
- [-inkey] private key to use with req
- [-keyout] file to output key to
- [-subj] use a specified subject name, e.g., O=wolfSSL/C=US/ST=WA/L=Seattle/CN=wolfSSL/OU=org-unit
- [-verify] check the signature on the req
- [-text] output human readable text of req
- [-noout] do not print out the generated results

Example:

```
wolfssl ecparam -genkey -out ecc.key -name secp384r1

wolfssl req -new -x509 -days 3650 -config selfsigned.conf -key ecc.key -out
   ecc.cert -outform der -sha256

wolfssl req -newkey rsa:2048 -keyout mykey.pem -out myreq.csr -subj "O=wolfSSL
   /C=US/CN=test"
```

1.4.23 Typical Workflow: RSA Key to Certificate

Step 1: Generate RSA key pair (see GENKEY Command)

```
wolfssl genkey rsa -size 2048 -out server -outform pem -output KEYPAIR
```

Step 2: Create CSR from private key

```
wolfssl req -new -key server.priv -out server.csr -subj "/C=US/ST=Washington/L
   =Seattle/O=wolfSSL/CN=example.com"
```

Step 3a: Create self-signed certificate

```
wolfssl req -x509 -key server.priv -in server.csr -out server.pem -days 365
```

Step 3b: Or create CA-signed certificate (see CA Command)

First, create a CA certificate:

```
wolfssl genkey rsa -size 2048 -out ca -outform pem -output KEYPAIR
wolfssl req -new -x509 -key ca.priv -out ca.pem -days 3650 -subj "/C=US/ST=
   Washington/L=Seattle/O=wolfSSL/CN=wolfSSL CA"
```

Then sign the server CSR with the CA:

```
wolfssl ca -in server.csr -out server-signed.pem -keyfile ca.priv -cert ca.pem
    -days 365 -md sha256
```

Step 4: Verify the certificate (see VERIFY Command)

```
wolfssl verify -CAfile ca.pem server-signed.pem
```

1.4.24 RSA Command

Performs RSA operations including signing, verification, and key management. Can handle both DER and PEM format for input and output.

RSA Sign

```
wolfssl rsa -sign -inkey <priv_key> -in <filename> -out <filename>
```

RSA Verify with Private Key

```
wolfssl rsa -verify -inkey <priv_key> -sigfile <filename> -out <filename>
```

RSA Verify with Public Key

```
wolfssl rsa -verify -inkey <pub_key> -sigfile <filename> -out <filename> -
    pubin
```

Arguments:

- [-sign] sign the input data
- [-verify] verify the signature
- [-inkey] key file for signing/verification
- [-in] input file to sign or process
- [-out] file to write result to (defaults to stdout)
- [-sigfile] signature file for verification
- [-pubin] expect a public key input
- [-inform] PEM or DER input format (defaults to PEM)
- [-outform] PEM or DER output format (defaults to PEM)
- [-passin] password for PEM encrypted files
- [-noout] do not print the key out when set
- [-modulus] print out the RSA modulus (n value)

Example:

```
wolfssl rsa -sign -inkey private.pem -in data.txt -out signature.bin

wolfssl rsa -verify -inkey public.pem -sigfile signature.bin -out verified.txt
     -pubin
```

1.4.25 sha256, sha384, and sha512 commands

Each command can be used to create a hash of its type. sha256 generates a sha256 hash and so on. The commands accept input in the form of stdin or a specified input file.

Example :

```
wolfssl -sha384 file.txt

echo "hi" | wolfssl -sha384
```

1.4.26 S_CLIENT Command

Basic TLS client for testing connections.

Arguments:

- [-connect] <ip>:<port> or <[ipv6]>:<port>
- [-starttls] protocol for STARTTLS (e.g., smtp)

- [-CAfile] CA file name for verification
- [-verify_return_error] close connection on verification error
- [-disable_stdin_check] disable stdin check

IPv6 Examples:

- `-connect '[::1]:11111'`
- `-connect '[fe80::63:57c0:9b88:77ca%en0]:11111'`
- `-connect '[2001:4860:4860::8888]:443'`

Example:

```
wolfssl s_client -connect 127.0.0.1:11111

wolfssl s_client -connect example.com:443 -CAfile ./certs/ca-cert.pem

wolfssl s_client -connect '[::1]:11111'
```

1.4.27 S_SERVER Command

A simple SSL/TLS server for testing. Listens on a specified port and handles incoming SSL/TLS connections.

Arguments:

- [-port] Port number to listen on
- [-key] Private key file (PEM format)
- [-cert] Certificate file (PEM format)
- [-noVerify] Disable client certificate verification
- [-CAfile] CA certificate file for client verification (PEM format)
- [-version] SSL/TLS version: 0=SSLv3, 1=TLS1.0, 2=TLS1.1, 3=TLS1.2, 4=TLS1.3
- [-naccept] Number of connections to accept (default: 1)
- [-www] Send HTTP response in HTML format
- [-readyFile] Create a ready file for external monitoring

Usage Examples:

Start a simple TLS server on port 4433:

```
wolfssl s_server -port 4433 -cert server-cert.pem -key server-key.pem
```

Start server with client certificate verification:

```
wolfssl s_server -port 4433 -cert server-cert.pem -key server-key.pem -CAfile
   ca-cert.pem
```

Start server accepting multiple connections with HTTP response:

```
wolfssl s_server -port 4433 -cert server-cert.pem -key server-key.pem -naccept
    10 -www
```

1.4.28 VERIFY Command

Verifies an X509 certificate given a CA. The last argument passed into the command is the certificate file name to be verified. If the verification is successful then “OK” will be printed out to stdout. Otherwise an error value and reason will be printed out.

Arguments:

- [-CAfile] file name for CA to be used with verify

- [-untrusted] file name for intermediate certificate to be used in verification (only one -untrusted cert is currently supported)
- [-crl_check] if CRL checking should be used
- [-partial_chain] allow verification with intermediate CA

Example:

```
wolfssl verify -CAfile ./certs/ca-cert.pem ./certs/server-cert.pem

wolfssl verify -CAfile ./certs/ca-cert.pem -untrusted ./certs/intermediate.pem
    ./certs/server-cert.pem
```

1.4.29 Typical Workflow: Verify CA-Signed Certificate

After creating a CA-signed certificate (see CA Command), verify it:

```
wolfssl verify -CAfile ca.pem server-signed.pem
```

Expected output on success:

```
verifying certificate file server-signed.pem
using CA file ca.pem
OK
```

1.4.30 VERSION Command

Displays version information for wolfCLU and the linked wolfSSL library.

Usage:

```
wolfssl version
```

Example Output:

```
You are using version 0.1.9 of the wolfssl Command Line Utility.
Linked to wolfSSL version 5.8.4
```

1.4.31 X509 Command

This command is used for parsing and printing out certificates.

Arguments:

- [-in] X509 file input
- [-inform] pem or der format for input (defaults to pem)
- [-out] file to output to
- [-outform] pem or der format for output (defaults to pem)
- [-req] input file is a CSR file
- [-signkey] a key for signing
- [-*] supported digests for signing (e.g., -sha256)
- [-extfile] config file
- [-extensions] section of the config file to use
- [-noout] no output
- [-subject] print out the subject name
- [-issuer] print out the issuer name
- [-serial] print out the serial number in hex
- [-dates] print out the valid dates of cert

- [-email] print out the subject name's email address
- [-fingerprint] print out the hash of the certificate in DER form
- [-purpose] print out the certificate's purpose
- [-hash] print out the hash of the certificate subject name
- [-text] print human readable text of X509
- [-modulus] print out the RSA key modulus
- [-pubkey] print out the public key

Example:

```
wolfssl x509 -in ./certs/server-cert.pem -text

wolfssl x509 -inform pem -in certs/ca-cert.pem -outform der -out certs/ca-cert
    -converted.der
```

1.4.32 Typical Workflow: View Certificate Details

After creating a certificate (see REQ Command or CA Command), view its details:

```
wolfssl x509 -in server.pem -text -noout
```

View specific information:

```
wolfssl x509 -in server.pem -subject -issuer -dates -noout
```

1.4.33 BASE64 Command

Encodes or decodes data using base64 encoding. By default, data is encoded to base64. Use the -d option to decode base64 data.

- [-in] input file to read from (default stdin)
- [-out] output file to write to (default stdout)
- [-d] decode the input data instead of encoding

Example for encoding:

```
wolfssl base64 -in plain_file.txt -out encoded_file.txt
```

Example for decoding:

```
wolfssl base64 -d -in encoded_file.txt -out decoded_file.txt
```

1.4.34 Configuration File Format

wolfCLU supports OpenSSL-style configuration files for the `req` and `ca` commands. This page describes the supported directives.

1.5 Basic Structure

Configuration files use INI-style format with sections and key-value pairs:

```
[section_name]
key = value
```

1.6 REQ Command Configuration

1.6.1 Main Section

The [req] section contains the main configuration:

Directive	Description
prompt	Set to no to disable interactive prompting
default_bits	Default key size in bits (e.g., 2048)
default_md	Default hash algorithm (e.g., sha256)
default_keyfile	Default key file path
distinguished_name	Section name containing DN fields
attributes	Section name containing attributes
x509_extensions	Section name containing X509 extensions

1.6.2 Distinguished Name Section

The distinguished name section (referenced by distinguished_name) defines certificate subject fields:

Directive	Description
countryName	Country code (2 letters, e.g., US)
stateOrProvinceName	State or province name
localityName	City or locality name
organizationName	Organization name
organizationalUnitName	Organizational unit name
commonName or CN	Common name (e.g., domain name)
emailAddress	Email address
name	Name
surname	Surname
givenName	Given name
initials	Initials
dnQualifier	DN qualifier

Each field can have optional modifiers:

Modifier	Description
<field>_default	Default value if not provided
<field>_min	Minimum length
<field>_max	Maximum length

1.6.3 Attributes Section

The attributes section (referenced by attributes) defines PKCS#9 attributes:

Directive	Description
challengePassword	PKCS#9 challenge password
unstructuredName	PKCS#9 unstructured name

1.7 X509 Extensions Section

The extensions section (referenced by `x509_extensions`) defines certificate extensions:

Directive	Description
`basicConstraints`	CA constraint and path length
`subjectKeyIdentifier`	Subject key identifier
`authorityKeyIdentifier`	Authority key identifier
`keyUsage`	Key usage flags
`subjectAltName`	Subject alternative names

1.7.1 basicConstraints

Defines whether the certificate is a CA and the maximum path length.

Format: `[critical, ]CA:TRUE|FALSE[, pathlen:<n>]`

Value	Description
`critical`	Mark as critical extension
`CA:TRUE`	This is a CA certificate
`CA:FALSE`	This is not a CA certificate
`pathlen:<n>`	Maximum certification path length

Examples:

```
basicConstraints = critical, CA:TRUE, pathlen:0
basicConstraints = CA:FALSE
```

1.7.2 keyUsage

Defines the purpose of the key contained in the certificate.

Format: `[critical, ]<usage>[, <usage>...]`

Value	Description
`digitalSignature`	Digital signature
`nonRepudiation`	Non-repudiation (also `contentCommitment`)
`keyEncipherment`	Key encipherment
`dataEncipherment`	Data encipherment
`keyAgreement`	Key agreement
`keyCertSign`	Certificate signing
`cRLSign`	CRL signing
`encipherOnly`	Encipher only (with keyAgreement)
`decipherOnly`	Decipher only (with keyAgreement)

Examples:

```
keyUsage = critical, digitalSignature, keyEncipherment
keyUsage = keyCertSign, cRLSign
```

1.7.3 subjectKeyIdentifier

Identifies the public key in the certificate.

Value	Description
hash	Use hash of public key

Example:

```
subjectKeyIdentifier = hash
```

1.7.4 subjectAltName

Specifies additional identities for the certificate subject. Use `@section_name` to reference a section containing the alternative names.

Example:

```
subjectAltName = @alt_names
```

1.7.4.1 Alternative Names Section

Directive	Description
`DNS.<n>`	DNS name (e.g., `DNS.1 = example.com`)
`IP.<n>`	IP address (e.g., `IP.1 = 192.168.1.1`)
`URI.<n>`	URI (e.g., `URI.1 = https://example.com`)
`email.<n>`	Email address (e.g., `email.1 = admin@example.com`)
`RID.<n>`	Registered ID / OID

Example:

```
[alt_names]
DNS.1 = example.com
DNS.2 = www.example.com
IP.1 = 192.168.1.1
email.1 = admin@example.com
```

1.8 CA Command Configuration

1.8.1 Main Section

The `[ca]` section specifies the default CA section:

```
[ca]
default_ca = CA_default
```

1.8.2 CA Section

The CA section (e.g., `[CA_default]`) contains CA-specific settings:

Directive	Description
serial	File containing serial number (hex format)
new_certs_dir	Directory for new certificates
certificate	CA certificate file
private_key	CA private key file
default_days	Default validity period in days
default_md	Default hash algorithm
x509_extensions	Section name for extensions
policy	Section name for DN policy
database	Certificate database file
unique_subject	Require unique subjects
crl_dir	Directory for CRLs
crl	CRL file
RANDFILE	Random seed file

1.9 Complete Examples

1.9.1 Server Certificate Request

```
[req]
prompt = no
default_bits = 2048
default_md = sha256
distinguished_name = req_dn
x509_extensions = v3_server

[req_dn]
countryName = US
stateOrProvinceName = Washington
localityName = Seattle
organizationName = wolfSSL
commonName = example.com
emailAddress = info@example.com

[v3_server]
basicConstraints = critical, CA:FALSE
keyUsage = digitalSignature, keyEncipherment
subjectKeyIdentifier = hash
subjectAltName = @alt_names

[alt_names]
DNS.1 = example.com
DNS.2 = www.example.com
IP.1 = 192.168.1.1
```

Usage:

```
wolfssl req -new -config server.conf -key server.priv -out server.csr
wolfssl req -new -x509 -config server.conf -key server.priv -out server.pem -
    days 365
```

1.9.2 CA Certificate

```
[req]
prompt = no
default_bits = 4096
default_md = sha256
distinguished_name = ca_dn
x509_extensions = v3_ca

[ca_dn]
countryName = US
stateOrProvinceName = Washington
localityName = Seattle
organizationName = wolfSSL
commonName = wolfSSL CA

[v3_ca]
basicConstraints = critical, CA:TRUE, pathlen:1
keyUsage = critical, keyCertSign, cRLSign
subjectKeyIdentifier = hash
```

Usage:

```
wolfssl req -new -x509 -config ca.conf -key ca.priv -out ca.pem -days 3650
```

1.9.3 Simple CSR (Minimal)

```
[req]
prompt = no
distinguished_name = req_dn

[req_dn]
commonName = myserver
```

Usage:

```
wolfssl req -new -config simple.conf -key server.priv -out server.csr
```

1.10 Serial Number File

The `serial` directive specifies a file containing the serial number in hexadecimal format:

```
01
```

The serial number is incremented after each certificate is signed. Without a configuration file specifying a serial file, wolfCLU generates a random serial number for each signed certificate.