Build and Install

This document describes installation on all supported operating systems (the Unix/Linux family, including macOS), OpenVMS, and Windows).

Table of Contents

- Prerequisites
- Notational Conventions
- Quick Installation Guide
  - Building OpenSSL
  - Installing OpenSSL
- Configuration Options
  - API Level
  - Cross Compile Prefix
  - Build Type
  - Directories
  - Compiler Warnings
  - ZLib Flags
  - Seeding the Random Generator
  - Setting the FIPS HMAC key
  - Enable and Disable Features
  - Displaying configuration data
- Installation Steps in Detail
  - Configure
  - Build
  - Test
  - Install
- Advanced Build Options
  - Environment Variables
  - Makefile Targets
  - Running Selected Tests
- Troubleshooting
  - Configuration Problems
  - Build Failures
  - Test Failures
- Notes
  - Notes on multi-threading
  - Notes on shared libraries
  - Notes on random number generation
  - Notes on assembler modules compilation

Prerequisites

To install OpenSSL, you will need:

- A “make” implementation
- Perl 5 with core modules (please read NOTES-PERL.md)
- The Perl module `Text::Template` (please read NOTES-PERL.md)
- an ANSI C compiler
- a development environment in the form of development libraries and C header files
- a supported operating system

For additional platform specific requirements, solutions to specific issues and other details, please read one of these:

- Notes for UNIX-like platforms
- Notes for Android platforms
- Notes for Windows platforms
- Notes for the DOS platform with DJGPP
- Notes for the OpenVMS platform
- Notes on Perl
- Notes on Valgrind

Notational conventions

Throughout this document, we use the following conventions.

Commands

Any line starting with a dollar sign is a command line.

```
$ command
```

The dollar sign indicates the shell prompt and is not to be entered as part of the command.

Choices

Several words in curly braces separated by pipe characters indicate a **mandatory choice**, to be replaced with one of the given words. For example, the line

```
$ echo { WORD1 | WORD2 | WORD3 }
```

represents one of the following three commands

```
$ echo WORD1
- or -
$ echo WORD2
- or -
$ echo WORD3
```

One or several words in square brackets separated by pipe characters denote an **optional choice**. It is similar to the mandatory choice, but it can also be omitted entirely.

So the line

```
$ echo [ WORD1 | WORD2 | WORD3 ]
```

represents one of the four commands

```
$ echo WORD1
- or -
$ echo WORD2
- or -
$ echo WORD3
- or -
$ echo
```

Arguments

Mandatory arguments are enclosed in double curly braces. A simple example would be

```
$ type {{ filename }}
```

which is to be understood to use the command `type` on some file name determined by the user.

Optional Arguments are enclosed in double square brackets.

```
[[ options ]]
```

Note that the notation assumes spaces around `{`, `}`, `[`, `]`, `{{`, `}}` and `[[`, `]]`. This is to differentiate from OpenVMS directory specifications, which also use [and], but without spaces.

Quick Installation Guide

If you just want to get OpenSSL installed without bothering too much about the details, here is the short version of how to build and install OpenSSL. If any of the following steps fails, please consult the Installation in Detail section below.

Building OpenSSL

Use the following commands to configure, build and test OpenSSL. The testing is optional, but recommended if you intend to install OpenSSL for production use.

Unix / Linux / macOS

```
$ ./Configure
$ make
$ make test
```

OpenVMS

Use the following commands to build OpenSSL:

```
$ perl Configure
$ mms
$ mms test
```

Windows

If you are using Visual Studio, open a Developer Command Prompt and issue the following commands to build OpenSSL.

```
$ perl Configure
$ nmake
$ nmake test
```

As mentioned in the Choices section, you need to pick one of the four Configure targets in the first command.

Most likely you will be using the `VC-WIN64A` target for 64bit Windows binaries (AMD64) or `VC-WIN32` for 32bit Windows binaries (X86). The other two options are `VC-WIN64I` (Intel IA64, Itanium) and `VC-CE` (Windows CE) are rather uncommon nowadays.

Installing OpenSSL

The following commands will install OpenSSL to a default system location.

Danger Zone: even if you are impatient, please read the following two paragraphs carefully before you install OpenSSL.

For security reasons the default system location is by default not writable for unprivileged users. So for the final installation step administrative privileges are required. The default system location and the procedure to obtain administrative privileges depends on the operating system. It is recommended to compile and test OpenSSL with normal user privileges and use administrative privileges only for the final installation step.

On some platforms OpenSSL is preinstalled as part of the Operating System. In this case it is highly recommended not to overwrite the system versions, because other applications or libraries might depend on it. To avoid breaking other applications, install your copy of OpenSSL to a different location which is not in the global search path for system libraries.

Finally, if you plan on using the FIPS module, you need to read the Post-installation Notes further down.

Unix / Linux / macOS

Depending on your distribution, you need to run the following command as root user or prepend `sudo` to the command:

```
$ make install
```

By default, OpenSSL will be installed to

```
/usr/local
```

More precisely, the files will be installed into the subdirectories

```
/usr/local/bin
/usr/local/lib
/usr/local/include
...
```

depending on the file type, as it is custom on Unix-like operating systems.

OpenVMS

Use the following command to install OpenSSL.

```
$ mms install
```

By default, OpenSSL will be installed to

```
SYS$COMMON:[OPENSSL]
```

Windows

If you are using Visual Studio, open the Developer Command Prompt *elevated* and issue the following command.

```
$ nmake install
```

The easiest way to elevate the Command Prompt is to press and hold down the both the `<CTRL>` and `<SHIFT>` key while clicking the menu item in the task menu.

The default installation location is

```
C:\Program Files\OpenSSL
```

for native binaries, or

```
C:\Program Files (x86)\OpenSSL
```

for 32bit binaries on 64bit Windows (WOW64).

Installing to a different location To install OpenSSL to a different location (for example into your home directory for testing purposes) run `Configure` as shown in the following examples.

The options `--prefix` and `--openssldir` are explained in further detail in Directories below, and the values used here are mere examples.

On Unix:

```
$ ./Configure --prefix=/opt/openssl --openssldir=/usr/local/ssl
```

On OpenVMS:

```
$ perl Configure --prefix=PROGRAM:[INSTALLS] --openssldir=SYS$MANAGER:[OPENSSL]
```

Note: if you do add options to the configuration command, please make sure you've read more than just this Quick Start, such as relevant `NOTES-*` files, the options outline below, as configuration options may change the outcome in otherwise unexpected ways.

Configuration Options

There are several options to `./Configure` to customize the build (note that for Windows, the defaults for `--prefix` and `--openssldir` depend on what configuration is used and what Windows implementation OpenSSL is built on. For more information, see the Notes for Windows platforms.

API Level

```
--api=x.y[.z]
```

Build the OpenSSL libraries to support the API for the specified version. If no-deprecated is also given, don't build with support for deprecated APIs in or below the specified version number. For example, adding

```
--api=1.1.0 no-deprecated
```

will remove support for all APIs that were deprecated in OpenSSL version 1.1.0 or below. This is a rather specialized option for developers. If you just intend to remove all deprecated APIs up to the current version entirely, just specify no-deprecated. If `--api` isn't given, it defaults to the current (minor) OpenSSL version.

Cross Compile Prefix

```
--cross-compile-prefix=<PREFIX>
```

The `<PREFIX>` to include in front of commands for your toolchain.

It is likely to have to end with dash, e.g. `a-b-c-` would invoke GNU compiler as `a-b-c-gcc`, etc. Unfortunately cross-compiling is too case-specific to put together one-size-fits-all instructions. You might have to pass more flags or set up environment variables to actually make it work. Android and iOS cases are discussed in corresponding `Configurations/15-*.conf` files. But there are cases when this option alone is sufficient. For example to build the mingw64 target on Linux `--cross-compile-prefix=x86_64-w64-mingw32-` works. Naturally provided that mingw packages are installed. Today Debian and Ubuntu users have option to install a number of prepackaged cross-compilers along with corresponding run-time and development packages for “alien” hardware. To give another example `--cross-compile-prefix=mipsel-linux-gnu-` suffices in such case.

For cross compilation, you must configure manually. Also, note that `--openssldir` refers to target’s file system, not one you are building on.

Build Type

```
--debug
```

Build OpenSSL with debugging symbols and zero optimization level.

```
--release
```

Build OpenSSL without debugging symbols. This is the default.

Directories

libdir

```
--libdir=DIR
```

The name of the directory under the top of the installation directory tree (see the `--prefix` option) where libraries will be installed. By default this is `lib`. Note that on Windows only static libraries (`*.lib`) will be stored in this location. Shared libraries (`*.dll`) will always be installed to the `bin` directory.

Some build targets have a multilib postfix set in the build configuration. For these targets the default libdir is `lib<multilib-postfix>`. Please use `--libdir=lib` to override the libdir if adding the postfix is undesirable.

openssldir

```
--openssldir=DIR
```

Directory for OpenSSL configuration files, and also the default certificate and key store. Defaults are:

```
Unix:           /usr/local/ssl
Windows:        C:\Program Files\Common Files\SSL
OpenVMS:        SYS$COMMON:[OPENSSL-COMMON]
```

For 32bit Windows applications on Windows 64bit (WOW64), always replace `C:\Program Files` by `C:\Program Files (x86)`.

prefix

```
--prefix=DIR
```

The top of the installation directory tree. Defaults are:

```
Unix:           /usr/local
Windows:        C:\Program Files\OpenSSL
OpenVMS:        SYS$COMMON:[OPENSSL]
```

Compiler Warnings

```
--strict-warnings
```

This is a developer flag that switches on various compiler options recommended for OpenSSL development. It only works when using gcc or clang as the compiler. If you are developing a patch for OpenSSL then it is recommended that you use this option where possible.

ZLib Flags

with-zlib-include

```
--with-zlib-include=DIR
```

The directory for the location of the zlib include file. This option is only necessary if zlib is used and the include file is not already on the system include path.

with-zlib-lib

```
--with-zlib-lib=LIB
```

On Unix: this is the directory containing the zlib library. If not provided the system library path will be used.

On Windows: this is the filename of the zlib library (with or without a path). This flag must be provided if the zlib-dynamic option is not also used. If `zlib-dynamic` is used then this flag is optional and defaults to `ZLIB1` if not provided.

On VMS: this is the filename of the zlib library (with or without a path). This flag is optional and if not provided then `GNV$LIBZSHR`, `GNV$LIBZSHR32` or `GNV$LIBZSHR64` is used by default depending on the pointer size chosen.

Seeding the Random Generator

```
--with-rand-seed=seed1[,seed2,...]
```

A comma separated list of seeding methods which will be tried by OpenSSL in order to obtain random input (a.k.a “entropy”) for seeding its cryptographically secure random number generator (CSPRNG). The current seeding methods are:

os

Use a trusted operating system entropy source. This is the default method if such an entropy source exists.

getrandom

Use the getrandom(2) or equivalent system call.

devrandom

Use the first device from the `DEVRANDOM` list which can be opened to read random bytes. The `DEVRANDOM` preprocessor constant expands to

```
"/dev/urandom","/dev/random","/dev/srandom"
```

on most unix-ish operating systems.

egd

Check for an entropy generating daemon. This source is ignored by the FIPS provider.

rdcpu

Use the `RDSEED` or `RDRAND` command if provided by the CPU.

librandom

Use librandom (not implemented yet). This source is ignored by the FIPS provider.

none

Disable automatic seeding. This is the default on some operating systems where no suitable entropy source exists, or no support for it is implemented yet. This option is ignored by the FIPS provider.

For more information, see the section Notes on random number generation at the end of this document.

Setting the FIPS HMAC key

```
--fips-key=value
```

As part of its self-test validation, the FIPS module must verify itself by performing a SHA-256 HMAC computation on itself. The default key is the SHA256 value of “the holy handgrenade of antioch” and is sufficient for meeting the FIPS requirements.

To change the key to a different value, use this flag. The value should be a hex string no more than 64 characters.

Enable and Disable Features

Feature options always come in pairs, an option to enable feature `xxxx`, and an option to disable it:

```
[ enable-xxxx | no-xxxx ]
```

Whether a feature is enabled or disabled by default, depends on the feature. In the following list, always the non-default variant is documented: if feature `xxxx` is disabled by default then `enable-xxxx` is documented and if feature `xxxx` is enabled by default then `no-xxxx` is documented.

no-afalgeng

Don’t build the AFALG engine.

This option will be forced on a platform that does not support AFALG.

enable-ktls

Build with Kernel TLS support.

This option will enable the use of the Kernel TLS data-path, which can improve performance and allow for the use of sendfile and splice system calls on TLS sockets. The Kernel may use TLS accelerators if any are available on the system. This option will be forced off on systems that do not support the Kernel TLS data-path.

enable-asan

Build with the Address sanitiser.

This is a developer option only. It may not work on all platforms and should never be used in production environments. It will only work when used with gcc or clang and should be used in conjunction with the no-shared option.

enable-acvp-tests

Build support for Automated Cryptographic Validation Protocol (ACVP) tests.

This is required for FIPS validation purposes. Certain ACVP tests require access to algorithm internals that are not normally accessible. Additional information related to ACVP can be found at https://github.com/usnistgov/ACVP.

no-asm

Do not use assembler code.

This should be viewed as debugging/troubleshooting option rather than for production use. On some platforms a small amount of assembler code may still be used even with this option.

no-async

Do not build support for async operations.

no-autoalginit

Don't automatically load all supported ciphers and digests.

Typically OpenSSL will make available all of its supported ciphers and digests. For a statically linked application this may be undesirable if small executable size is an objective. This only affects libcrypto. Ciphers and digests will have to be loaded manually using `EVP_add_cipher()` and `EVP_add_digest()` if this option is used. This option will force a non-shared build.

no-autoerrinit

Don't automatically load all libcrypto/libssl error strings.

Typically OpenSSL will automatically load human readable error strings. For a statically linked application this may be undesirable if small executable size is an objective.

no-autoload-config

Don't automatically load the default `openssl.cnf` file.

Typically OpenSSL will automatically load a system config file which configures default SSL options.

enable-buildtest-c++

While testing, generate C++ buildtest files that simply check that the public OpenSSL header files are usable standalone with C++.

Enabling this option demands extra care. For any compiler flag given directly as configuration option, you must ensure that it's valid for both the C and the C++ compiler. If not, the C++ build test will most likely break. As an alternative, you can use the language specific variables, `CFLAGS` and `CXXFLAGS`.

–banner=text

Use the specified text instead of the default banner at the end of configuration.

–w

On platforms where the choice of 32-bit or 64-bit architecture is not explicitly specified, `Configure` will print a warning message and wait for a few seconds to let you interrupt the configuration. Using this flag skips the wait.

no-bulk

Build only some minimal set of features. This is a developer option used internally for CI build tests of the project.

no-cached-fetch

Never cache algorithms when they are fetched from a provider. Normally, a provider indicates if the algorithms it supplies can be cached or not. Using this option will reduce run-time memory usage but it also introduces a significant performance penalty. This option is primarily designed to help with detecting incorrect reference counting.

no-capieng

Don't build the CAPI engine.

This option will be forced if on a platform that does not support CAPI.

no-cmp

Don't build support for Certificate Management Protocol (CMP) and Certificate Request Message Format (CRMF).

no-cms

Don't build support for Cryptographic Message Syntax (CMS).

no-comp

Don't build support for SSL/TLS compression.

If this option is enabled (the default), then compression will only work if the zlib or `zlib-dynamic` options are also chosen.

enable-crypto-mdebug

This now only enables the `failed-malloc` feature.

enable-crypto-mdebug-backtrace

This is a no-op; the project uses the compiler's address/leak sanitizer instead.

no-ct

Don't build support for Certificate Transparency (CT).

no-deprecated

Don't build with support for deprecated APIs up until and including the version given with `--api` (or the current version, if `--api` wasn't specified).

no-dgram

Don't build support for datagram based BIOs.

Selecting this option will also force the disabling of DTLS.

no-dso

Don't build support for loading Dynamic Shared Objects (DSO)

enable-devcryptoeng

Build the `/dev/crypto` engine.

This option is automatically selected on the BSD platform, in which case it can be disabled with `no-devcryptoeng`.

no-dynamic-engine

Don't build the dynamically loaded engines.

This only has an effect in a shared build.

no-ec

Don't build support for Elliptic Curves.

no-ec2m

Don't build support for binary Elliptic Curves

enable-ec_nistp_64_gcc_128

Enable support for optimised implementations of some commonly used NIST elliptic curves.

This option is only supported on platforms:

- with little-endian storage of non-byte types
- that tolerate misaligned memory references
- where the compiler:
    - supports the non-standard type `__uint128_t`

  - defines the built-in macro `__SIZEOF_INT128__`

enable-egd

Build support for gathering entropy from the Entropy Gathering Daemon (EGD).

no-engine

Don't build support for loading engines.

no-err

Don't compile in any error strings.

enable-external-tests

Enable building of integration with external test suites.

This is a developer option and may not work on all platforms. The following external test suites are currently supported:

- GOST engine test suite
- Python PYCA/Cryptography test suite
- krb5 test suite

See the file test/README-external.md for further details.

no-filenames

Don't compile in filename and line number information (e.g. for errors and memory allocation).

enable-fips

Build (and install) the FIPS provider

no-fips-securitychecks

Don't perform FIPS module run-time checks related to enforcement of security parameters such as minimum security strength of keys.

enable-fuzz-libfuzzer, enable-fuzz-afl

Build with support for fuzzing using either libfuzzer or AFL.

These are developer options only. They may not work on all platforms and should never be used in production environments.

See the file fuzz/README.md for further details.

no-gost

Don't build support for GOST based ciphersuites.

Note that if this feature is enabled then GOST ciphersuites are only available if the GOST algorithms are also available through loading an externally supplied engine.

no-legacy

Don't build the legacy provider.

Disabling this also disables the legacy algorithms: MD2 (already disabled by default).

no-makedepend

Don't generate dependencies.

no-module

Don't build any dynamically loadable engines.

This also implies `no-dynamic-engine`.

no-multiblock

Don't build support for writing multiple records in one go in libssl

Note: this is a different capability to the pipelining functionality.

no-nextprotoneg

Don't build support for the Next Protocol Negotiation (NPN) TLS extension.

no-ocsp

Don't build support for Online Certificate Status Protocol (OCSP).

no-padlockeng

Don't build the padlock engine.

no-hw-padlock

As synonym for `no-padlockeng`. Deprecated and should not be used.

no-pic

Don't build with support for Position Independent Code.

no-pinshared

Don't pin the shared libraries.

By default OpenSSL will attempt to stay in memory until the process exits. This is so that libcrypto and libssl can be properly cleaned up automatically via an `atexit()` handler. The handler is registered by libcrypto and cleans up both libraries. On some platforms the `atexit()` handler will run on unload of libcrypto (if it has been dynamically loaded) rather than at process exit. This option can be used to stop OpenSSL from attempting to stay in memory until the process exits. This could lead to crashes if either libcrypto or libssl have already been unloaded at the point that the atexit handler is invoked, e.g. on a platform which calls `atexit()` on unload of the library, and libssl is unloaded before libcrypto then a crash is likely to happen. Applications can suppress running of the `atexit()` handler at run time by using the `OPENSSL_INIT_NO_ATEXIT` option to `OPENSSL_init_crypto()`. See the man page for it for further details.

no-posix-io

Don't use POSIX IO capabilities.

no-psk

Don't build support for Pre-Shared Key based ciphersuites.

no-quic

Don't build support for QUIC API from BoringSSL.

no-rdrand

Don't use hardware RDRAND capabilities.

no-rfc3779

Don't build support for RFC3779, "X.509 Extensions for IP Addresses and AS Identifiers".

sctp

Build support for Stream Control Transmission Protocol (SCTP).

no-shared

Do not create shared libraries, only static ones.

See Notes on shared libraries below.

no-sock

Don't build support for socket BIOs.

no-srp

Don't build support for Secure Remote Password (SRP) protocol or SRP based ciphersuites.

no-srtp

Don't build Secure Real-Time Transport Protocol (SRTP) support.

no-sse2

Exclude SSE2 code paths from 32-bit x86 assembly modules.

Normally SSE2 extension is detected at run-time, but the decision whether or not the machine code will be executed is taken solely on CPU capability vector. This means that if you happen to run OS kernel which does not support SSE2 extension on Intel P4 processor, then your application might be exposed to "illegal instruction" exception. There might be a way to enable support in kernel, e.g. FreeBSD kernel can be compiled with `CPU_ENABLE_SSE`, and there is a way to disengage SSE2 code paths upon application start-up, but if you aim for wider "audience" running such kernel, consider `no-sse2`. Both the `386` and `no-asm` options imply `no-sse2`.

no-ssl-trace

Don't build with SSL Trace capabilities.

This removes the `-trace` option from `s_client` and `s_server`, and omits the `SSL_trace()` function from libssl.

Disabling `ssl-trace` may provide a small reduction in libssl binary size.

no-static-engine

Don't build the statically linked engines.

This only has an impact when not built "shared".

no-stdio

Don't use anything from the C header file `stdio.h` that makes use of the `FILE` type. Only libcrypto and libssl can be built in this way. Using this option will suppress building the command line applications. Additionally, since the OpenSSL tests also use the command line applications, the tests will also be skipped.

no-tests

Don't build test programs or run any tests.

no-threads

Don't build with support for multi-threaded applications.

threads

Build with support for multi-threaded applications. Most platforms will enable this by default. However, if on a platform where this is not the case then this will usually require additional system-dependent options!

See Notes on multi-threading below.

enable-trace

Build with support for the integrated tracing api.

See manual pages OSSL_trace_set_channel(3) and OSSL_trace_enabled(3) for details.

no-ts

Don't build Time Stamping (TS) Authority support.

enable-ubsan

Build with the Undefined Behaviour sanitiser (UBSAN).

This is a developer option only. It may not work on all platforms and should never be used in production environments. It will only work when used with gcc or clang and should be used in conjunction with the `-DPEDANTIC` option (or the `--strict-warnings` option).

no-ui-console

Don't build with the User Interface (UI) console method

The User Interface console method enables text based console prompts.

enable-unit-test

Enable additional unit test APIs.

This should not typically be used in production deployments.

no-uplink

Don't build support for UPLINK interface.

enable-weak-ssl-ciphers

Build support for SSL/TLS ciphers that are considered "weak"

Enabling this includes for example the RC4 based ciphersuites.

zlib

Build with support for zlib compression/decompression.

zlib-dynamic

Like the zlib option, but has OpenSSL load the zlib library dynamically when needed.

This is only supported on systems where loading of shared libraries is supported.

386

In 32-bit x86 builds, use the 80386 instruction set only in assembly modules

The default x86 code is more efficient, but requires at least an 486 processor. Note: This doesn't affect compiler generated code, so this option needs to be accompanied by a corresponding compiler-specific option.

no-{protocol}

```
no-{ssl|ssl3|tls|tls1|tls1_1|tls1_2|tls1_3|dtls|dtls1|dtls1_2}
```

Don't build support for negotiating the specified SSL/TLS protocol.

If `no-tls` is selected then all of `tls1`, `tls1_1`, `tls1_2` and `tls1_3` are disabled. Similarly `no-dtls` will disable `dtls1` and `dtls1_2`. The `no-ssl` option is synonymous with `no-ssl3`. Note this only affects version negotiation. OpenSSL will still provide the methods for applications to explicitly select the individual protocol versions.

no-{protocol}-method

```
no-{ssl|ssl3|tls|tls1|tls1_1|tls1_2|tls1_3|dtls|dtls1|dtls1_2}-method
```

Analogous to `no-{protocol}` but in addition do not build the methods for applications to explicitly select individual protocol versions. Note that there is no `no-tls1_3-method` option because there is no application method for TLSv1.3.

Using individual protocol methods directly is deprecated. Applications should use `TLS_method()` instead.

enable-{algorithm}

```
enable-{md2|rc5}
```

Build with support for the specified algorithm.

no-{algorithm}

```
no-{aria|bf|blake2|camellia|cast|chacha|cmac|
    des|dh|dsa|ecdh|ecdsa|idea|md4|mdc2|ocb|
    poly1305|rc2|rc4|rmd160|scrypt|seed|
    siphash|siv|sm2|sm3|sm4|whirlpool}
```

Build without support for the specified algorithm.

The `ripemd` algorithm is deprecated and if used is synonymous with `rmd160`.

Compiler-specific options

```
-Dxxx, -Ixxx, -Wp, -lxxx, -Lxxx, -Wl, -rpath, -R, -framework, -static
```

These system specific options will be recognised and passed through to the compiler to allow you to define preprocessor symbols, specify additional libraries, library directories or other compiler options. It might be worth noting that some compilers generate code specifically for processor the compiler currently executes on. This is not necessarily what you might have in mind, since it might be unsuitable for execution on other, typically older, processor. Consult your compiler documentation.

Take note of the Environment Variables documentation below and how these flags interact with those variables.

```
-xxx, +xxx, /xxx
```

Additional options that are not otherwise recognised are passed through as they are to the compiler as well. Unix-style options beginning with a `-` or `+` and Windows-style options beginning with a `/` are recognized. Again, consult your compiler documentation.

If the option contains arguments separated by spaces, then the URL-style notation `%20` can be used for the space character in order to avoid having to quote the option. For example, `-opt%20arg` gets expanded to `-opt arg`. In fact, any ASCII character can be encoded as %xx using its hexadecimal encoding.

Take note of the Environment Variables documentation below and how these flags interact with those variables.

Environment Variables

```
VAR=value
```

Assign the given value to the environment variable VAR for Configure.

These work just like normal environment variable assignments, but are supported on all platforms and are confined to the configuration scripts only. These assignments override the corresponding value in the inherited environment, if there is one.

The following variables are used as "make variables" and can be used as an alternative to giving preprocessor, compiler and linker options directly as configuration. The following variables are supported:

```
AR              The static library archiver.
ARFLAGS         Flags for the static library archiver.
AS              The assembler compiler.
ASFLAGS         Flags for the assembler compiler.
CC              The C compiler.
CFLAGS          Flags for the C compiler.
CXX             The C++ compiler.
CXXFLAGS        Flags for the C++ compiler.
CPP             The C/C++ preprocessor.
CPPFLAGS        Flags for the C/C++ preprocessor.
CPPDEFINES      List of CPP macro definitions, separated
                by a platform specific character (':' or
                space for Unix, ';' for Windows, ',' for
                VMS).  This can be used instead of using
                -D (or what corresponds to that on your
                compiler) in CPPFLAGS.
CPPINCLUDES     List of CPP inclusion directories, separated
                the same way as for CPPDEFINES.  This can
                be used instead of -I (or what corresponds
                to that on your compiler) in CPPFLAGS.
HASHBANGPERL    Perl invocation to be inserted after '#!'
                in public perl scripts (only relevant on
                Unix).
LD              The program linker (not used on Unix, $(CC)
                is used there).
LDFLAGS         Flags for the shared library, DSO and
                program linker.
LDLIBS          Extra libraries to use when linking.
                Takes the form of a space separated list
                of library specifications on Unix and
                Windows, and as a comma separated list of
                libraries on VMS.
RANLIB          The library archive indexer.
RC              The Windows resource compiler.
RCFLAGS         Flags for the Windows resource compiler.
RM              The command to remove files and directories.
```

These cannot be mixed with compiling/linking flags given on the command line. In other words, something like this isn't permitted.

```
$ ./Configure -DFOO CPPFLAGS=-DBAR -DCOOKIE
```

Backward compatibility note:

To be compatible with older configuration scripts, the environment variables are ignored if compiling/linking flags are given on the command line, except for the following:

```
AR, CC, CXX, CROSS_COMPILE, HASHBANGPERL, PERL, RANLIB, RC, and WINDRES
```

For example, the following command will not see `-DBAR`:

```
$ CPPFLAGS=-DBAR ./Configure -DCOOKIE
```

However, the following will see both set variables:

```
$ CC=gcc CROSS_COMPILE=x86_64-w64-mingw32- ./Configure -DCOOKIE
```

If `CC` is set, it is advisable to also set `CXX` to ensure both the C and C++ compiler are in the same "family". This becomes relevant with `enable-external-tests` and `enable-buildtest-c++`.

Reconfigure

```
reconf
reconfigure
```

Reconfigure from earlier data.

This fetches the previous command line options and environment from data saved in `configdata.pm` and runs the configuration process again, using these options and environment. Note: NO other option is permitted together with `reconf`. Note: The original configuration saves away values for ALL environment variables that were used, and if they weren't defined, they are still saved away with information that they weren't originally defined. This information takes precedence over environment variables that are defined when reconfiguring.

Displaying configuration data

The configuration script itself will say very little, and finishes by creating `configdata.pm`. This perl module can be loaded by other scripts to find all the configuration data, and it can also be used as a script to display all sorts of configuration data in a human readable form.

For more information, please do:

```
$ ./configdata.pm --help                         # Unix
```

or

```
$ perl configdata.pm --help                      # Windows and VMS
```

Installation Steps in Detail

Configure OpenSSL

Automatic Configuration

In previous version, the `config` script determined the platform type and compiler and then called `Configure`. Starting with this release, they are the same.

Unix / Linux / macOS

```
$ ./Configure [[ options ]]
```

OpenVMS

```
$ perl Configure [[ options ]]
```

Windows

```
$ perl Configure [[ options ]]
```

Manual Configuration

OpenSSL knows about a range of different operating system, hardware and compiler combinations. To see the ones it knows about, run

```
$ ./Configure LIST                               # Unix
```

or

```
$ perl Configure LIST                            # All other platforms
```

For the remainder of this text, the Unix form will be used in all examples. Please use the appropriate form for your platform.

Pick a suitable name from the list that matches your system. For most operating systems there is a choice between using cc or gcc. When you have identified your system (and if necessary compiler) use this name as the argument to `Configure`. For example, a `linux-elf` user would run:

```
$ ./Configure linux-elf [[ options ]]
```

Creating your own Configuration

If your system isn't listed, you will have to create a configuration file named `Configurations/{{ something }}.conf` and add the correct configuration for your system. See the available configs as examples and read Configurations/README.md and Configurations/README-design.md for more information.

The generic configurations `cc` or `gcc` should usually work on 32 bit Unix-like systems.

`Configure` creates a build file (`Makefile` on Unix, `makefile` on Windows and `descrip.mms` on OpenVMS) from a suitable template in `Configurations/`, and defines various macros in `include/openssl/configuration.h` (generated from `include/openssl/configuration.h.in`.

Out of Tree Builds

OpenSSL can be configured to build in a build directory separate from the source code directory. It's done by placing yourself in some other directory and invoking the configuration commands from there.

Unix example

```
$ mkdir /var/tmp/openssl-build
$ cd /var/tmp/openssl-build
$ /PATH/TO/OPENSSL/SOURCE/Configure [[ options ]]
```

OpenVMS example

```
$ set default sys$login:
$ create/dir [.tmp.openssl-build]
$ set default [.tmp.openssl-build]
$ perl D:[PATH.TO.OPENSSL.SOURCE]Configure [[ options ]]
```

Windows example

```
$ C:
$ mkdir \temp-openssl
$ cd \temp-openssl
$ perl d:\PATH\TO\OPENSSL\SOURCE\Configure [[ options ]]
```

Paths can be relative just as well as absolute. `Configure` will do its best to translate them to relative paths whenever possible.

Build OpenSSL

Build OpenSSL by running:

```
$ make                                              # Unix
$ mms                                               ! (or mmk) OpenVMS
$ nmake                                             # Windows
```

This will build the OpenSSL libraries (`libcrypto.a` and `libssl.a` on Unix, corresponding on other platforms) and the OpenSSL binary (`openssl`). The libraries will be built in the top-level directory, and the binary will be in the `apps/` subdirectory.

If the build fails, take a look at the Build Failures subsection of the Troubleshooting section.

Test OpenSSL

After a successful build, and before installing, the libraries should be tested. Run:

```
$ make test                                      # Unix
$ mms test                                       ! OpenVMS
$ nmake test                                     # Windows
```

Warning: you MUST run the tests from an unprivileged account (or disable your privileges temporarily if your platform allows it).

See test/README.md for further details how run tests.

See test/README-dev.md for guidelines on adding tests.

Install OpenSSL

If everything tests ok, install OpenSSL with

```
$ make install                                   # Unix
$ mms install                                    ! OpenVMS
$ nmake install                                  # Windows
```

Note that in order to perform the install step above you need to have appropriate permissions to write to the installation directory.

The above commands will install all the software components in this directory tree under `<PREFIX>` (the directory given with `--prefix` or its default):

Unix / Linux / macOS

```
bin/           Contains the openssl binary and a few other
               utility scripts.
include/openssl
               Contains the header files needed if you want
               to build your own programs that use libcrypto
               or libssl.
lib            Contains the OpenSSL library files.
lib/engines    Contains the OpenSSL dynamically loadable engines.

share/man/man1 Contains the OpenSSL command line man-pages.
share/man/man3 Contains the OpenSSL library calls man-pages.
share/man/man5 Contains the OpenSSL configuration format man-pages.
share/man/man7 Contains the OpenSSL other misc man-pages.

share/doc/openssl/html/man1
```

```
share/doc/openssl/html/man3
share/doc/openssl/html/man5
share/doc/openssl/html/man7
               Contains the HTML rendition of the man-pages.
```

OpenVMS

'arch' is replaced with the architecture name, `ALPHA` or `IA64`, 'sover' is replaced with the shared library version (`0101` for 1.1), and 'pz' is replaced with the pointer size OpenSSL was built with:

```
[.EXE.'arch']  Contains the openssl binary.
[.EXE]         Contains a few utility scripts.
[.include.openssl]
               Contains the header files needed if you want
               to build your own programs that use libcrypto
               or libssl.
[.LIB.'arch']  Contains the OpenSSL library files.
[.ENGINES'sover''pz'.'arch']
               Contains the OpenSSL dynamically loadable engines.
[.SYS$STARTUP] Contains startup, login and shutdown scripts.
               These define appropriate logical names and
               command symbols.
[.SYSTEST]     Contains the installation verification procedure.
[.HTML]        Contains the HTML rendition of the manual pages.
```

Additional Directories

Additionally, install will add the following directories under OPENSSLDIR (the directory given with `--openssldir` or its default) for you convenience:

```
certs          Initially empty, this is the default location
               for certificate files.
private        Initially empty, this is the default location
               for private key files.
misc           Various scripts.
```

The installation directory should be appropriately protected to ensure unprivileged users cannot make changes to OpenSSL binaries or files, or install engines. If you already have a pre-installed version of OpenSSL as part of your Operating System it is recommended that you do not overwrite the system version and instead install to somewhere else.

Package builders who want to configure the library for standard locations, but have the package installed somewhere else so that it can easily be packaged, can use

```
$ make DESTDIR=/tmp/package-root install         # Unix
$ mms/macro="DESTDIR=TMP:[PACKAGE-ROOT]" install ! OpenVMS
```

The specified destination directory will be prepended to all installation target paths.

Compatibility issues with previous OpenSSL versions

COMPILING existing applications

Starting with version 1.1.0, OpenSSL hides a number of structures that were previously open. This includes all internal libssl structures and a number of EVP types. Accessor functions have been added to allow controlled access to the structures' data.

This means that some software needs to be rewritten to adapt to the new ways of doing things. This often amounts to allocating an instance of a structure explicitly where you could previously allocate them on the stack as automatic variables, and using the provided accessor functions where you would previously access a structure's field directly.

Some APIs have changed as well. However, older APIs have been preserved when possible.

Post-installation Notes

With the default OpenSSL installation comes a FIPS provider module, which needs some post-installation attention, without which it will not be usable. This involves using the following command:

```
$ openssl fipsinstall
```

See the openssl-fipsinstall(1) manual for details and examples.

Advanced Build Options

Environment Variables

A number of environment variables can be used to provide additional control over the build process. Typically these should be defined prior to running `Configure`. Not all environment variables are relevant to all platforms.

```
AR
               The name of the ar executable to use.

BUILDFILE
               Use a different build file name than the platform default
               ("Makefile" on Unix-like platforms, "makefile" on native Windows,
               "descrip.mms" on OpenVMS).  This requires that there is a
               corresponding build file template.
               See [Configurations/README.md](Configurations/README.md)
               for further information.
```

```
CC
               The compiler to use. Configure will attempt to pick a default
               compiler for your platform but this choice can be overridden
               using this variable. Set it to the compiler executable you wish
               to use, e.g. gcc or clang.

CROSS_COMPILE
               This environment variable has the same meaning as for the
               "--cross-compile-prefix" Configure flag described above. If both
               are set then the Configure flag takes precedence.

HASHBANGPERL
               The command string for the Perl executable to insert in the
               #! line of perl scripts that will be publicly installed.
               Default: /usr/bin/env perl
               Note: the value of this variable is added to the same scripts
               on all platforms, but it's only relevant on Unix-like platforms.

KERNEL_BITS
               This can be the value `32` or `64` to specify the architecture
               when it is not "obvious" to the configuration. It should generally
               not be necessary to specify this environment variable.

NM
               The name of the nm executable to use.

OPENSSL_LOCAL_CONFIG_DIR
               OpenSSL comes with a database of information about how it
               should be built on different platforms as well as build file
               templates for those platforms. The database is comprised of
               ".conf" files in the Configurations directory.  The build
               file templates reside there as well as ".tmpl" files. See the
               file [Configurations/README.md](Configurations/README.md)
               for further information about the format of ".conf" files
               as well as information on the ".tmpl" files.
               In addition to the standard ".conf" and ".tmpl" files, it is
               possible to create your own ".conf" and ".tmpl" files and
               store them locally, outside the OpenSSL source tree.
               This environment variable can be set to the directory where
               these files are held and will be considered by Configure
               before it looks in the standard directories.

PERL
               The name of the Perl executable to use when building OpenSSL.
               Only needed if builing should use a different Perl executable
               than what is used to run the Configure script.

RANLIB
               The name of the ranlib executable to use.

RC
               The name of the rc executable to use. The default will be as
               defined for the target platform in the ".conf" file. If not
               defined then "windres" will be used. The WINDRES environment
               variable is synonymous to this. If both are defined then RC
               takes precedence.

WINDRES
               See RC.
```

Makefile Targets

The `Configure` script generates a Makefile in a format relevant to the specific platform. The Makefiles provide a number of targets that can be used. Not all targets may be available on all platforms. Only the most common targets are described here. Examine the Makefiles themselves for the full list.

```
all
               The target to build all the software components and
               documentation.

build_sw
               Build all the software components.
               THIS IS THE DEFAULT TARGET.

build_docs
               Build all documentation components.

clean
               Remove all build artefacts and return the directory to a "clean"
               state.

depend
               Rebuild the dependencies in the Makefiles. This is a legacy
               option that no longer needs to be used since OpenSSL 1.1.0.

install
               Install all OpenSSL components.

install_sw
               Only install the OpenSSL software components.
```

```
install_docs
               Only install the OpenSSL documentation components.

install_man_docs
               Only install the OpenSSL man pages (Unix only).

install_html_docs
               Only install the OpenSSL HTML documentation.

install_fips
               Install the FIPS provider module configuration file.

list-tests
               Prints a list of all the self test names.

test
               Build and run the OpenSSL self tests.

uninstall
               Uninstall all OpenSSL components.

reconfigure
reconf
               Re-run the configuration process, as exactly as the last time
               as possible.

update
               This is a developer option. If you are developing a patch for
               OpenSSL you may need to use this if you want to update
               automatically generated files; add new error codes or add new
               (or change the visibility of) public API functions. (Unix only).
```

Running Selected Tests

You can specify a set of tests to be performed using the `make` variable `TESTS`.

See the section Running Selected Tests of test/README.md.

Troubleshooting

Configuration Problems

Selecting the correct target

The `./Configure` script tries hard to guess your operating system, but in some cases it does not succeed. You will see a message like the following:

```
$ ./Configure
Operating system: x86-whatever-minix
This system (minix) is not supported. See file INSTALL.md for details.
```

Even if the automatic target selection by the `./Configure` script fails, chances are that you still might find a suitable target in the `Configurations` directory, which you can supply to the `./Configure` command, possibly after some adjustment.

The `Configurations/` directory contains a lot of examples of such targets. The main configuration file is 10-main.conf, which contains all targets that are officially supported by the OpenSSL team. Other configuration files contain targets contributed by other OpenSSL users. The list of targets can be found in a Perl list `my %targets = ( ... )`.

```
my %targets = (
...
"target-name" => {
    inherit_from     => [ "base-target" ],
    CC               => "...",
    cflags           => add("..."),
    asm_arch         => '...',
    perlasm_scheme   => "...",
},
...
)
```

If you call `./Configure` without arguments, it will give you a list of all known targets. Using `grep`, you can lookup the target definition in the `Configurations/` directory. For example the `android-x86_64` can be found in Configurations/15-android.conf.

The directory contains two README files, which explain the general syntax and design of the configuration files.

- Configurations/README.md
- Configurations/README-design.md

If you need further help, try to search the openssl-users mailing list or the GitHub Issues for existing solutions. If you don't find anything, you can raise an issue to ask a question yourself.

More about our support resources can be found in the SUPPORT file.

Configuration Errors

If the `./Configure` or `./Configure` command fails with an error message, read the error message carefully and try to figure out whether you made a mistake (e.g., by providing a wrong option), or whether the script is working incorrectly. If you think you encountered a bug, please raise an issue on GitHub to file a bug report.

Along with a short description of the bug, please provide the complete configure command line and the relevant output including the error message.

Note: To make the output readable, pleace add a 'code fence' (three backquotes ``` on a separate line) before and after your output:

~~~
 ```
 ./Configure [your arguments...]

 [output...]

 ```
~~~

Build Failures

If the build fails, look carefully at the output. Try to locate and understand the error message. It might be that the compiler is already telling you exactly what you need to do to fix your problem.

There may be reasons for the failure that aren't problems in OpenSSL itself, for example if the compiler reports missing standard or third party headers.

If the build succeeded previously, but fails after a source or configuration change, it might be helpful to clean the build tree before attempting another build. Use this command:

```
$ make clean                                     # Unix
$ mms clean                                      ! (or mmk) OpenVMS
$ nmake clean                                    # Windows
```

Assembler error messages can sometimes be sidestepped by using the `no-asm` configuration option. See also notes.

Compiling parts of OpenSSL with gcc and others with the system compiler will result in unresolved symbols on some systems.

If you are still having problems, try to search the openssl-users mailing list or the GitHub Issues for existing solutions. If you think you encountered an OpenSSL bug, please raise an issue to file a bug report. Please take the time to review the existing issues first; maybe the bug was already reported or has already been fixed.

Test Failures

If some tests fail, look at the output. There may be reasons for the failure that isn't a problem in OpenSSL itself (like an OS malfunction or a Perl issue).

You may want increased verbosity, that can be accomplished as described in section Test Failures of test/README.md.

You may also want to selectively specify which test(s) to perform. This can be done using the `make` variable `TESTS` as described in section Running Selected Tests of test/README.md.

If you find a problem with OpenSSL itself, try removing any compiler optimization flags from the `CFLAGS` line in the Makefile and run `make clean; make` or corresponding.

To report a bug please open an issue on GitHub, at https://github.com/openssl/openssl/issues.

Notes

Notes on multi-threading

For some systems, the OpenSSL `Configure` script knows what compiler options are needed to generate a library that is suitable for multi-threaded applications. On these systems, support for multi-threading is enabled by default; use the `no-threads` option to disable (this should never be necessary).

On other systems, to enable support for multi-threading, you will have to specify at least two options: `threads`, and a system-dependent option. (The latter is `-D_REENTRANT` on various systems.) The default in this case, obviously, is not to include support for multi-threading (but you can still use `no-threads` to suppress an annoying warning message from the `Configure` script.)

OpenSSL provides built-in support for two threading models: pthreads (found on most UNIX/Linux systems), and Windows threads. No other threading models are supported. If your platform does not provide pthreads or Windows threads then you should use `Configure` with the `no-threads` option.

For pthreads, all locks are non-recursive. In addition, in a debug build, the mutex attribute `PTHREAD_MUTEX_ERRORCHECK` is used. If this is not available on your platform, you might have to add `-DOPENSSL_NO_MUTEX_ERRORCHECK` to your `Configure` invocation. (On Linux `PTHREAD_MUTEX_ERRORCHECK` is an enum value, so a built-in ifdef test cannot be used.)

Notes on shared libraries

For most systems the OpenSSL `Configure` script knows what is needed to build shared libraries for libcrypto and libssl. On these systems the shared libraries will be created by default. This can be suppressed and only static libraries created by using the `no-shared` option. On systems where OpenSSL does not know how to build shared libraries the `no-shared` option will be forced and only static libraries will be created.

Shared libraries are named a little differently on different platforms. One way or another, they all have the major OpenSSL version number as part of the file name, i.e. for OpenSSL 1.1.x, `1.1` is somehow part of the name.

On most POSIX platforms, shared libraries are named `libcrypto.so.1.1` and `libssl.so.1.1`.

on Cygwin, shared libraries are named `cygcrypto-1.1.dll` and `cygssl-1.1.dll` with import libraries `libcrypto.dll.a` and `libssl.dll.a`.

On Windows build with MSVC or using MingW, shared libraries are named `libcrypto-1_1.dll` and `libssl-1_1.dll` for 32-bit Windows, `libcrypto-1_1-x64.dll` and `libssl-1_1-x64.dll` for 64-bit x86_64 Windows, and `libcrypto-1_1-ia64.dll` and `libssl-1_1-ia64.dll` for IA64 Windows. With MSVC, the import libraries are named `libcrypto.lib` and `libssl.lib`, while with MingW, they are named `libcrypto.dll.a` and `libssl.dll.a`.

On VMS, shareable images (VMS speak for shared libraries) are named `ossl$libcrypto0101_shr.exe` and `ossl$libssl0101_shr.exe`. However, when OpenSSL is specifically built for 32-bit pointers, the shareable images are named `ossl$libcrypto0101_shr32.exe` and `ossl$libssl0101_shr32.exe` instead, and when built for 64-bit pointers, they are named `ossl$libcrypto0101_shr64.exe` and `ossl$libssl0101_shr64.exe`.

Notes on random number generation

Availability of cryptographically secure random numbers is required for secret key generation. OpenSSL provides several options to seed the internal CSPRNG. If not properly seeded, the internal CSPRNG will refuse to deliver random bytes and a "PRNG not seeded error" will occur.

The seeding method can be configured using the `--with-rand-seed` option, which can be used to specify a comma separated list of seed methods. However, in most cases OpenSSL will choose a suitable default method, so it is not necessary to explicitly provide this option. Note also that not all methods are available on all platforms. The FIPS provider will silently ignore seed sources that were not validated.

I) On operating systems which provide a suitable randomness source (in form of a system call or system device), OpenSSL will use the optimal available method to seed the CSPRNG from the operating system's randomness sources. This corresponds to the option `--with-rand-seed=os`.

II) On systems without such a suitable randomness source, automatic seeding and reseeding is disabled (`--with-rand-seed=none`) and it may be necessary to install additional support software to obtain a random seed and reseed the CSPRNG manually. Please check out the manual pages for `RAND_add()`, `RAND_bytes()`, `RAND_egd()`, and the FAQ for more information.

Notes on assembler modules compilation

Compilation of some code paths in assembler modules might depend on whether the current assembler version supports certain ISA extensions or not. Code paths that use the AES-NI, PCLMULQDQ, SSSE3, and SHA extensions are always assembled. Apart from that, the minimum requirements for the assembler versions are shown in the table below:

ISA extension	GNU as	nasm	llvm
AVX	2.19	2.09	3.0
AVX2	2.22	2.10	3.1
ADCX/ADOX	2.23	2.10	3.3
AVX512	2.25	2.11.8	3.6 (*)
AVX512IFMA	2.26	2.11.8	6.0 (*)
VAES	2.30	2.13.3	6.0 (*)

(*) Even though AVX512 support was implemented in llvm 3.6, prior to version 7.0 an explicit -march flag was apparently required to compile assembly modules. But then the compiler generates processor-specific code, which in turn contradicts the idea of performing dispatch at run-time, which is facilitated by the special variable `OPENSSL_ia32cap`. For versions older than 7.0, it is possible to work around the problem by forcing the build procedure to use the following script:

```
#!/bin/sh
exec clang -no-integrated-as "$@"
```

instead of the real clang. In which case it doesn't matter what clang version is used, as it is the version of the GNU assembler that will be checked.
