Notes for Windows platforms

There are various options to build and run OpenSSL on the Windows platforms.

"Native" OpenSSL uses the Windows APIs directly at run time. To build a native OpenSSL you can either use:

```
Microsoft Visual C++ (MSVC) C compiler on the command line
```

or Embarcadero C++Builder or MinGW cross compiler run on the GNU-like development environment MSYS2 or run on Linux or Cygwin

"Hosted" OpenSSL relies on an external POSIX compatibility layer for building (using GNU/Unix shell, compiler, and tools) and at run time. For this option you can use Cygwin.

Native builds using Visual C++

The native builds using Visual C++ have a VC-* prefix.

Requirement details

In addition to the requirements and instructions listed in INSTALL.md, these are required as well:

Perl

We recommend Strawberry Perl, available from http://strawberryperl.com/ Please read NOTES.PERL for more information, including the use of CPAN. An alternative is ActiveState Perl, https://www.activestate.com/ActivePerl for which you may need to explicitly build the Perl module Win32/Console.pm via https://platform.activestate.com/ActiveState and then download it.

Microsoft Visual C compiler.

Since these are proprietary and ever-changing we cannot test them all. Older versions may not work. Use a recent version wherever possible.

Netwide Assembler (NASM)

NASM is the only supported assembler. It is available from https://www.nasm.us.

Quick start

1. Install Perl
2. Install NASM
3. Make sure both Perl and NASM are on your %PATH%
4. Use Visual Studio Developer Command Prompt with administrative privileges, choosing one of its variants depending on the intended architecture. Or run "cmd" and execute "vcvarsall.bat" with one of the options

x86, x86_amd64, x86_arm, x86_arm64, amd64, amd64_x86, amd64_arm, or amd64_arm64. This sets up the environment variables needed for nmake.exe, cl.exe, etc. See also https://docs.microsoft.com/cpp/build/building-on-the-command-line

5. From the root of the OpenSSL source directory enter perl Configure VC-WIN32 if you want 32-bit OpenSSL or perl Configure VC-WIN64A if you want 64-bit OpenSSL or perl Configure to let Configure figure out the platform
6. nmake
7. nmake test
8. nmake install

For the full installation instructions, or if anything goes wrong at any stage, check the INSTALL.md file.

Installation directories

The default installation directories are derived from environment variables.

For VC-WIN32, the following defaults are use:

```
PREFIX:       %ProgramFiles(x86)%\OpenSSL
OPENSSLDIR:   %CommonProgramFiles(x86)%\SSL
```

For VC-WIN64, the following defaults are use:

```
PREFIX:       %ProgramW6432%\OpenSSL
OPENSSLDIR:   %CommonProgramW6432%\SSL
```

Should those environment variables not exist (on a pure Win32 installation for examples), these fallbacks are used:

```
PREFIX:       %ProgramFiles%\OpenSSL
OPENSSLDIR:   %CommonProgramFiles%\SSL
```

ALSO NOTE that those directories are usually write protected, even if your account is in the Administrators group. To work around that, start the command prompt by right-clicking on it and choosing "Run as Administrator" before running 'nmake install'. The other solution is, of course, to choose a different set of directories by using --prefix and --openssldir when configuring.

Special notes for Universal Windows Platform builds, aka VC-*-UWP

- UWP targets only support building the static and dynamic libraries.
- You should define the platform type to "uwp" and the target arch via "vcvarsall.bat" before you compile. For example, if you want to build "arm64" builds, you should run "vcvarsall.bat x86_arm64 uwp".

Native builds using Embarcadero C++Builder

This toolchain (a descendant of Turbo/Borland C++) is an alternative to MSVC. OpenSSL currently includes an experimental 32-bit configuration targeting the Clang-based compiler (bcc32c.exe) in v10.3.3 Community Edition. https://www.embarcadero.com/products/cbuilder/starter

1. Install Perl.
2. Open the RAD Studio Command Prompt.
3. Go to the root of the OpenSSL source directory and run: perl Configure BC-32 --prefix=%CD%
4. make -N
5. make -N test
6. Build your program against this OpenSSL:
    - Set your include search path to the "include" subdirectory of OpenSSL.
    - Set your library search path to the OpenSSL source directory.

Note that this is very experimental. Support for 64-bit and other Configure options is still pending.

Native builds using MinGW

MinGW offers an alternative way to build native OpenSSL, by cross compilation.

- Usually the build is done on Windows in a GNU-like environment called MSYS2.

  MSYS2 provides GNU tools, a Unix-like command prompt, and a UNIX compatibility layer for applications. However, in this context it is only used for building OpenSSL. The resulting OpenSSL does not rely on MSYS2 to run and is fully native.

  Requirement details

    - MSYS2 shell, from https://www.msys2.org/
    - Perl, at least version 5.10.0, which usually comes pre-installed with MSYS2
    - make, installed using "pacman -S make" into the MSYS2 environment
    - MinGW[64] compiler: mingw-w64-i686-gcc and/or mingw-w64-x86_64-gcc. These compilers must be on your MSYS2 $PATH. A common error is to not have these on your $PATH. The MSYS2 version of gcc will not work correctly here.

  In the MSYS2 shell do the configuration depending on the target architecture:

  ```
  ./Configure mingw ...
  ```

  or ./Configure mingw64 ... or ./Configure ...

  for the default architecture.

  Apart from that, follow the Unix / Linux instructions in INSTALL.md.

- It is also possible to build mingw[64] on Linux or Cygwin.

  In this case configure with the corresponding --cross-compile-prefix= option. For example

  ```
  ./Configure mingw --cross-compile-prefix=i686-w64-mingw32- ...
  ```

  or ./Configure mingw64 --cross-compile-prefix=x86_64-w64-mingw32- ... This requires that you've installed the necessary add-on packages for mingw[64] cross compilation.

Linking native applications

This section applies to all native builds.

If you link with static OpenSSL libraries then you're expected to additionally link your application with WS2_32.LIB, GDI32.LIB, ADVAPI32.LIB, CRYPT32.LIB and USER32.LIB. Those developing non-interactive service applications might feel concerned about linking with GDI32.LIB and USER32.LIB, as they are justly associated with interactive desktop, which is not available to service processes. The toolkit is designed to detect in which context it's currently executed, GUI, console app or service, and act accordingly, namely whether or not to actually make GUI calls. Additionally those who wish to /DELAYLOAD:GDI32.DLL and /DELAYLOAD:USER32.DLL and actually keep them off service process should consider implementing and exporting from .exe image in question own _OPENSSL_isservice not relying on USER32.DLL. E.g., on Windows Vista and later you could:

```
__declspec(dllexport) __cdecl BOOL _OPENSSL_isservice(void)
{
    DWORD sess;

    if (ProcessIdToSessionId(GetCurrentProcessId(), &sess))
        return sess == 0;
    return FALSE;
}
```

If you link with OpenSSL .DLLs, then you're expected to include into your application code a small "shim" snippet, which provides the glue between the OpenSSL BIO layer and your compiler run-time. See also the OPENSSL_Applink manual page.

Hosted builds using Cygwin

Cygwin implements a POSIX/Unix runtime system (cygwin1.dll) on top of the Windows subsystem and provides a Bash shell and GNU tools environment. Consequently, a build of OpenSSL with Cygwin is virtually identical to the Unix procedure.

To build OpenSSL using Cygwin, you need to:

- Install Cygwin, see https://cygwin.com/
- Install Cygwin Perl, at least version 5.10.0 and ensure it is in the $PATH
- Run the Cygwin Bash shell

Apart from that, follow the Unix / Linux instructions in INSTALL.md.

NOTE: "make test" and normal file operations may fail in directories mounted as text (i.e. mount -t c:\somewhere /home) due to Cygwin stripping of carriage returns. To avoid this ensure that a binary mount is used, e.g. mount -b c:\somewhere /home.