Botan Build Guide

Jack Lloyd
lloyd@randombit.net

Contents

1 Introduction **2**

2 For the Impatient **2**

3 Building the Library **2**
3.1 POSIX / Unix 4
3.2 MS Windows 4
3.3 Configuration Parameters 6
3.4 Multiple Builds 6
3.5 Local Configuration 6

4 Modules **7**

5 Building Applications **8**
5.1 Unix 8
5.2 MS Windows 8

1 Introduction

This document describes how to build Botan on Unix/POSIX and MS Windows systems. The POSIX oriented descriptions should apply to most common Unix systems (including MacOS X), along with POSIX-ish systems like BeOS, QNX, and Plan 9. Currently, systems other than Windows and POSIX (such as VMS, MacOS 9, OS/390, OS/400, ...) are not supported by the build system, primarily due to lack of access. Please contact the maintainer if you would like to build Botan on such a system.

2 For the Impatient

```
$ ./configure.pl
$ make
$ make install
```

Or using `nmake`, if you're compiling on Windows with Visual C++. On platforms that do not understand the '#!' convention for beginning script files, or that have Perl installed in an unusual spot, you might need to prefix the `configure.pl` command with `perl` or `/path/to/perl`.

The autoconfiguaration abilities of `configure.pl` were only recently added, so they may break if you run it on something unusual. In addition, you are certain to get more features, and possibly better optimization, by explicitly specifying how you want to library configured. How to do this is detailed below. Also, if you don't want to use the default compiler (typically either GNU C++ or Visual C++, depending on the platform), you will need to specify one.

3 Building the Library

The first step is to run `configure.pl`, which is a Perl script that creates various directories, config files, and a Makefile for building everything. It is run as `./configure.pl CC-OS-CPU <extra args>`. The script requires at least Perl 5.6; any later version should also work.

The tuple CC-OS-CPU specifies what system Botan is being built for, in terms of the C++ compiler, the operating system, and the CPU model. For example, to use GNU C++ on a FreeBSD box that has an Alpha EV6 CPU, one would use "gcc-freebsd-alphaev6", and for Visual C++ on Windows with a Pentium II, "msvc-windows-pentium2". To get the list of values for `CC`, `OS`, and `CPU` that `configure.pl` supports, run it with the "`--help`" option.

You can put basically anything reasonable for CPU: the script knows about a large number of different architectures, their sub-models, and common aliases for them. The script does not display all the possibilities in its help message because there are simply too many entries. You should only select the 64-bit version of a CPU (such as "sparc64" or "mips64") if your operating system knows how to handle 64-bit object code – a 32-bit kernel on a 64-bit CPU will generally not like 64-bit code. For example, gcc-solaris-sparc64 will not work unless you're running a 64-bit Solaris kernel (for 32-bit Solaris running on an UltraSPARC system, you want gcc-solaris-sparc32-v9). You may or may not have to install 64-bit versions of libc and related system libraries as well.

The script also knows about the various extension modules available. You can enable one or more with the option "`--modules=MOD`", where `MOD` is some name that identifies the extension (or a comma separated list of them). Modules provide additional capabilities which require the use of APIs not provided by ISO C/C++.

Not all OSes or CPUs have specific support in `configure.pl`. If the CPU architecture of your system isn't supported by `configure.pl`, use 'generic'. This setting disables machine-specific optimization flags. Similarly, setting OS to 'generic' disables things which depend greatly on OS support (specifically, shared libraries).

However, it's impossible to guess which options to give to a system compiler. Thus, if you want to compile Botan with a compiler which `configure.pl` does not support, you will need to tell it how that compiler works. This is done by adding a new file in the directory `misc/config/cc`; the existing files should put you in the right direction.

The script tries to guess what kind of makefile to generate, and it almost always guesses correctly (basically, Visual C++ uses NMAKE with Windows commands, and everything else uses Unix make with POSIX commands). Just in case, you can override it with `--make-style=somestyle`. The styles Botan currently knows about are 'unix' (normal Unix makefiles), and 'nmake', the make variant commonly used by Windows compilers. To add a new variant (eg, a build script for VMS), you will need to create a new template file in `misc/config/makefile`.

3.1 POSIX / Unix

The basic build procedure on Unix and Unix-like systems is:

```
$ ./configure.pl CC-OS-CPU --module-set=[unix|beos] --modules=<other mods>
$ make
# You may need to set your LD_LIBRARY_PATH or equivalent for ./check to run
$ make check # optional, but a good idea
$ make install
```

The 'unix' module set should work on most POSIX/Unix systems out there (including MacOS X), while the 'beos' module is specific to BeOS. While the two sets share a number of modules, some normal Unix ones don't work on BeOS (in particular, BeOS doesn't have a working **mmap** function), and BeOS has a few extras just for it. The library will pick a default module set for you based on the value of OS, so there is rarely a reason to specify that.

The `make install` target has a default directory in which it will install Botan (on everything that's a real Unix, it's `/usr/local`). You can override this by using the `--prefix` argument to `configure.pl`, like so:

```
./configure.pl --prefix=/opt <other arguments>
```

On Unix, the makefile has to decide who should own the files once they are installed. By default, it uses `root:root`, but on some systems (for example, MacOS X), there is no `root` group. Also, if you don't have root access on the system you will want them to be installed owned by something other than root (like yourself). You can override the defaults at install time by setting the `OWNER` and `GROUP` variables from the command line.

```
make OWNER=lloyd GROUP=users install
```

On some systems shared libraries might not be immediately visible to the runtime linker. For example, on Linux you may have to edit `/etc/ld.so.conf` and run `ldconfig` (as root) in order for new shared libraries to be picked up by the linker. An alternative is to set your `LD_LIBRARY_PATH` shell variable to include the directory that the Botan libraries were installed into.

3.2 MS Windows

The situation is not much different here. We'll assume you're using Visual C++ (for Cygwin, the Unix instructions are probably more relevant). You need to have a copy of Perl installed, and have both Perl and Visual C++ in your path.

```
> perl configure.pl msvc-windows-<CPU> --module-set=win32
> nmake
> nmake check # optional, but recommended
```

By default, the configure script will include the 'win32' module set for you. This includes a pair of entropy sources for use on Windows; at some point in the future it will also add support for high-resolution timers, mutexes for thread safety, and other useful things.

For Win95 pre OSR2, the `es_capi` module will not work, because CryptoAPI didn't exist. All versions of NT4 lack the ToolHelp32 interface, which is how `es_win32` does its slow polls, so a version of the library built with that module will not load under NT4. Later systems (98/ME/2000/XP) support both methods, so this shouldn't be much of an issue.

Unfortunately, there currently isn't an install script usable on Windows. Basically all you have to do is copy the newly created `libbotan.lib` to someplace where you can find it later (say, `C:\Botan\`). Then

copy the entire `include\botan` directory, which was constructed when you built the library, into the same directory.

When building your applications, all you have to do is tell the compiler to look for both include files and library files in `C:\Botan`, and it will find both.

3.3 Configuration Parameters

There are some configuration parameters which you may want to tweak before building the library. These can be found in `config.h`. This file is overwritten every time the configure script is run (and does not exist until after you run the script for the first time).

Also included in `config.h` are macros which are defined if one or more extensions are available. All of them begin with `BOTAN_EXT_`. For example, if `BOTAN_EXT_COMPRESSOR_BZIP2` is defined, then an application using Botan can include `<botan/bzip2.h>` and use the Bzip2 filters.

`BOTAN_MP_WORD_BITS`: This macro controls the size of the words used for calculations with the MPI implementation in Botan. You can choose 8, 16, 32, or 64, with 32 being the default. You can use 8, 16, or 32 bit words on any CPU, but the value should be set to the same size as the CPU's registers for best performance. You can only use 64-bit words if an assembly module (such as `mp_ia32` or `mp_asm64`) is used. If the appropriate module is available, 64 bits are used, otherwise this is set to 32. Unless you are building for a 8 or 16-bit CPU, this isn't worth messing with.

`BOTAN_VECTOR_OVER_ALLOCATE`: The memory container `SecureVector` will over-allocate requests by this amount (in elements). In several areas of the library, we grow a vector fairly often. By over-allocating by a small amount, we don't have to do allocations as often (which is good, because the allocators can be quite slow). If you *really* want to reduce memory usage, set it to 0. Otherwise, the default should be perfectly fine.

`BOTAN_DEFAULT_BUFFER_SIZE`: This constant is used as the size of buffers throughout Botan. A good rule of thumb would be to use the page size of your machine. The default should be fine for most, if not all, purposes.

`BOTAN_GZIP_OS_CODE`: The OS code is included in the Gzip header when compressing. The default is 255, which means 'Unknown'. You can look in RFC 1952 for the full list; the most common are Windows (0) and Unix (3). There is also a Macintosh (7), but it probably makes more sense to use the Unix code on OS X.

3.4 Multiple Builds

It may be useful to run multiple builds with different configurations. Specify `--build-dir=<dir>` to set up a build environment in a different directory.

3.5 Local Configuration

You may want to do something peculiar with the configuration; to support this there is a flag to `configure.pl` called `--local-config=<file>`. The contents of the file are inserted into `build/build.h` which is (indirectly) included into every Botan header and source file.

4 Modules

There are a fairly large number of modules included with Botan. Some of these are extremely useful, while others are only necessary in very unusual circumstances. The modules included with this release are:

- **alloc_mmap**: Allocates memory using memory mappings of temporary files. This means that if the OS swaps all or part of the application, the sensitive data will be swapped to where we can later clean it, rather than somewhere in the swap partition.

  comp_bzip2: Enables an application to perform bzip2 compression and decompression using the library. Available on any system that has bzip2.

- **comp_zlib**: Enables an application to perform zlib compression and decompression using the library. Available on any system that has zlib.

- **eng_aep**: An engine that uses any available AEP accelerator card to speed up PK operations. You have to have the AEP drivers installed for this to link correctly, but you don't have to have a card installed - it will automatically be enabled if a card is detected at run time.

- **eng_gmp**: An engine that uses GNU MP to speed up PK operations. GNU MP 4.1 or later is required.

- **eng_ossl**: An engine that uses OpenSSL's BN library to speed up PK operations. OpenSSL 0.9.7 or later is required.

- **es_beos**: An entropy source that uses BeOS-specific APIs to gather (hopefully unpredictable) data from the system.

- **es_capi**: An entropy source that uses the Win32 CryptoAPI function `CryptGenRandom` to gather entropy. Supported on NT4, Win95 OSR2, and all later Windows systems.

- **es_egd**: An entropy source that accesses EGD (the entropy gathering daemon). Common on Unix systems that don't have `/dev/random`.

- **es_ftw**: Gather entropy by reading files from a particular file tree. Usually used with `/proc`; most other file trees don't have sufficient variability over time to be useful.

- **es_unix**: Gather entropy by running various Unix programs, like `arp` and `vmstat`, and reading their output in the hopes that at least some of it will be unpredictable to an attacker.

- **es_win32**: Gather entropy by walking through various pieces of information about processes running on the system. Does not run on NT4, but should run on all other Win32 systems.

- **fd_unix**: Let the users of `Pipe` perform I/O with Unix file descriptors in addition to `iostream` objects.

- **ml_unix**: Add hooks for locking memory into RAM. Usually requires the application to run as `root` to actually work, but if the application is not allowed to call `mlock`, no harm results.

- **mp_asm64**: Use inline assembly to access the multiply instruction available on some 64-bit CPUs. This module only runs on Alpha, AMD64, IA-64, MIPS64, and PowerPC-64. Typically PKI operations are several times as fast with this module than without.

- **mux_pthr**: Add support for using `pthread` mutexes to lock internal data structures. Important if you are using threads with the library.

- **mux_qt**: Add support for using Qt mutexes to lock internal data structures.

- **tm_hard**: Use the contents of the CPU cycle counter when generating random bits to further randomize the results. Works on x86 (Pentium and up), Alpha, and SPARCv9.

- **tm_posix**: Use the POSIX realtime clock as a high-resolution timer.

- **tm_unix**: Use the traditional Unix `gettimeofday` as a high resolution timer.

- **tm_win32**: Use Win32's `GetSystemTimeAsFileTime` as a high resolution timer.

5 Building Applications

5.1 Unix

Botan usually links in several different system libraries (such as `librt` and `libz`), depending on which modules are configured at compile time. In many environments, particularly ones using static libraries, an application has to link against the same libraries as Botan for the linking step to succeed. But how does it figure out what libraries it *is* linked against?

The answer is to ask the `botan-config` script. This basically solves the same problem all the other `*-config` scripts solve, and in basically the same manner. At some point in the future, a transition to `pkg-config` will be made (as it's less work, and has more features), but right now it doesn't exist on most Unix systems, while a plain Bourne shell script will run fine on anything.

There are 4 options:

`--prefix[=DIR]`: If no argument, print the prefix where Botan is installed (such as `/opt` or `/usr/local`). If an argument is specified, other options given with the same command will execute as if Botan as actually installed at `DIR` and not where it really is; or at least where `botan-config` thinks it really is. I should mention that it

`--version`: Print the Botan version number.

`--cflags`: Print options that should be passed to the compiler whenever a C++ file is compiled. Typically this is used for setting include paths.

`--libs`: Print options for which libraries to link to (this includes `-lbotan`).

Your `Makefile` can run `botan-config` and get the options necessary for getting your application to compile and link, regardless of whatever crazy libraries Botan might be linked against.

5.2 MS Windows

No special help exists for building applications on Windows. However, given that typically Windows software is distributed as binaries, this is less of a problem - only the developer needs to worry about it. As long as they can remember where they installed Botan, they just have to set the appropriate flags in their Makefile/project file.