Botan Build Guide

Jack Lloyd
lloyd@randombit.net

2010-06-10

Contents

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.

Botan's build is controlled by configure.py, which is a Python script. Python 2.5 or later is required. If you want to use the (incompatible) Python 3, you must first run the `2to3` script on it.

2 For the Impatient

```
$ ./configure.py [--prefix=/some/directory]
$ make
$ make check
$ 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 Python installed in an unusual spot, you might need to prefix the `configure.py` command with `python` or `/path/to/python`.

3 Building the Library

The first step is to run `configure.py`, which is a Python script that creates various directories, config files, and a Makefile for building everything. The script requires at least Python 2.5; any later version of Python 2.x should also work. Python 3.1 will also work but requires an extra step, see the section "Using Python 3.1", later in this document.

The script will attempt to guess what kind of system you are trying to compile for (and will print messages telling you what it guessed). You can override this process by passing the options `--cc`, `--os`, and `--cpu`.

You can pass basically anything reasonable with `--cpu`: the script knows about a large number of different architectures, their sub-models, and common aliases for them. 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.

By default the script tries to figure out what will work on your system, and use that. It will print a display at the end showing which algorithms have and have not been enabled. For instance on one system we might see lines like:

```
INFO: Skipping mod because CPU incompatible - asm_amd64 mp_amd64 mp_asm64 sha1_amd64
INFO: Skipping mod because OS incompatible - cryptoapi_rng win32_stats
INFO: Skipping mod because compiler incompatible - mp_ia32_msvc
INFO: Skipping mod because loaded on request only - bzip2 gnump openssl qt_mutex zlib
```

The ones that are 'loaded on request only' have to be explicitly asked for, because they rely on third party libraries which your system might not have. For instance to enable zlib support, add `--with-zlib` to your invocation of `configure.py`.

You can control which algorithms and modules are built using the options `--enable-modules=MODS` and `--disable-modules=MODS`, for instance `--enable-modules=zlib` and `--disable-modules=rc5,idea`. Modules not listed on the command line will simply be loaded if needed or if configured to load by default. If

you use `--no-autoload`, only the most core modules will be included; you can then explicitly enable things that you want to use with enable-modules. This is useful for creating a minimal build targetted to a specific application.

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 `src/build-data/makefile`.

3.1 POSIX / Unix

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

```
$ ./configure.py [--enable-modules=<list>] [--cc=CC]
$ 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
```

On Unix systems the script will default to using GCC; use `--cc` if you want something else. For instance use `--cc=icc` for Intel C++ and `--cc=clang` for Clang.

The `make install` target has a default directory in which it will install Botan (typically `/usr/local`). You can override this by using the `--prefix` argument to `configure.py`, like so:

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

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

If you don't want to deal with building botan on Windows, check the website; commonly prebuilt Windows binaries with installers are available, especially for stable versions.

You need to have a copy of Python installed, and have both Python and your chosen compiler in your path. Open a command shell (or the SDK shell), and run:

```
> python configure.py --cc=msvc (or --cc=gcc for MinGW) [--cpu=CPU]
> nmake
> nmake check # optional, but recommended
> nmake install
```

For Win95 pre OSR2, the `cryptoapi_rng` module will not work, because CryptoAPI didn't exist. And all versions of NT4 lack the ToolHelp32 interface, which is how `win32_stats` does its slow polls, so a version of the library built with that module will not load under NT4. Later versions of Windows support both methods, so this shouldn't be much of an issue anymore.

By default the install target will be 'C:\botan'; you can modify this with the `--prefix` option.

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. Or you can move them to a place where they will be in the default compiler search paths (consult your documentation and/or local expert for details).

4 Trickier Things

4.1 Modules Relying on Third Party Libraries

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. Most are loaded (or not) automatically as necessary, but some require external libraries are thus must be enabled at build time; these include:

- **bzip2**: Enables an application to perform bzip2 compression and decompression using the library. Available on any system that has bzip2. To enable, use option `--with-bzip2`

  zlib: Enables an application to perform zlib compression and decompression using the library. Available on any system that has zlib. To enable, use option `--with-zlib`

- **gnump**: An engine that uses GNU MP to speed up PK operations. GNU MP 4.1 or later is required. To enable, use option `--with-gnump`

- **openssl**: An engine that uses OpenSSL to speed up public key operations and some ciphers/hashes. OpenSSL 0.9.7 or later is required. Note that, unlike GNU MP, OpenSSL's versions are not always faster than the versions built into botan. To enable, use option `--with-openssl`

4.2 Amalgamation

You can also configure Botan to be built using only a single source file; this is quite convenient if you plan to embed the library into another application. To do so, run `configure.py` with whatever arguments you would ordinarily use, along with the option `--gen-amalgamation`. This will create two (rather large) files, `botan_all.h` and `botan_all.cpp`.

Whenever you would have included a botan header, you can then include `botan_all.h`, and include `botan_all.cpp` along with the rest of the source files in your build. If you want to be able to easily switch between amalgamated and non-amalgamated versions (for instance to take advantage of prepackaged versions of botan on operating systems that support it), you can instead ignore `botan_all.h` and use the headers from `build/include` as normal.

4.3 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.

4.4 Using Python 3.1

The versions of Python begininning with 3 are (intentionally) incompatible with the (currently more common) 2.x series. If you want to use Python 3.1 to set up the build, you'll have to use the `2to3` program (included in the Python distribution) on the script; this will convert the script to the Python 3.x dialect:

```
  $ python ./configure.py
  File "configure.py", line 860
    except KeyError, e:
                   ^
SyntaxError: invalid syntax
  $ # incompatible python version, let's fix it
  $ 2to3 -w configure.py
[...]
RefactoringTool: Files that were modified:
```

```
RefactoringTool: configure.py
  $ python ./configure.py
[...]
```

4.5 Local Configuration

You may want to do something peculiar with the configuration; to support this there is a flag to `configure.py` called `--with-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.

4.6 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 `build/build.h` are macros which are defined if one or more extensions are available. All of them begin with `BOTAN_HAS_`. For example, if `BOTAN_HAS_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.

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.

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.

Botan also by default installs a file for `pkg-config`, namespaced by the major and minor versions. So it can be used, for instance, as

```
$ pkg-config botan-1.9 --modversion
1.9.8
$ pkg-config botan-1.9 --cflags
-I/usr/local/include
$ pkg-config botan-1.9 --libs
-L/usr/local/lib -lbotan -lm -lbz2 -lpthread -lrt
```

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.

6 Language Wrappers

6.1 Building the Python wrappers

The Python wrappers for Botan use Boost.Python, so you must have Boost installed. To build the wrappers, add the flag

`--with-boost-python`

to `configure.py`. This will create a second makefile, `Makefile.python`, with instructions for building the Python module. After building the library, execute

```
$ make -f Makefile.python
```

to build the module. Currently only Unix systems are supported, and the Makefile assumes that the version of Python you want to build against is the same one you used to run `configure.py`.

To install the module, use the `install` target.

Examples of using the Python module can be seen in `doc/python`

6.2 Building the Perl XS wrappers

To build the Perl XS wrappers, change your directory to `src/wrap/perl-xs` and run `perl Makefile.PL`, then run `make` to build the module and `make test` to run the test suite.

```
$ perl Makefile.PL
Checking if your kit is complete...
```

```
Looks good
Writing Makefile for Botan
$ make
cp Botan.pm blib/lib/Botan.pm
AutoSplitting blib/lib/Botan.pm (blib/lib/auto/Botan)
/usr/bin/perl5.8.8 /usr/lib64/perl5/5.8.8/ExtUtils/xsubpp  [...]
g++ -c   -Wno-write-strings -fexceptions  -g   [...]
Running Mkbootstrap for Botan ()
chmod 644 Botan.bs
rm -f blib/arch/auto/Botan/Botan.so
g++  -shared Botan.o  -o blib/arch/auto/Botan/Botan.so  \
          -lbotan -lbz2 -lpthread -lrt -lz     \

chmod 755 blib/arch/auto/Botan/Botan.so
cp Botan.bs blib/arch/auto/Botan/Botan.bs
chmod 644 blib/arch/auto/Botan/Botan.bs
Manifying blib/man3/Botan.3pm
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl5.8.8 [...]
t/base64......ok
t/filt........ok
t/hex.........ok
t/oid.........ok
t/pipe........ok
t/x509cert....ok
All tests successful.
Files=6, Tests=83,  0 wallclock secs ( 0.08 cusr +  0.02 csys =  0.10 CPU)
```