Building Node.js

Depending on what platform or features you need, the build process may differ. After you've built a binary, running the test suite to confirm that the binary works as intended is a good next step.

If you can reproduce a test failure, search for it in the Node.js issue tracker or file a new issue.

Table of contents

- Supported platforms
  - Input
  - Strategy
  - Platform list
  - Supported toolchains
  - Official binary platforms and toolchains
    - OpenSSL asm support
  - Previous versions of this document
- Building Node.js on supported platforms
  - Note about Python
  - Unix and macOS
    - Unix prerequisites
    - macOS prerequisites
    - Building Node.js
    - Installing Node.js
    - Running Tests
    - Running Coverage
    - Building the documentation
    - Building a debug build
    - Building an ASAN build
    - Speeding up frequent rebuilds when developing
    - Troubleshooting Unix and macOS builds
  - Windows
    - Prerequisites
      - Option 1: Manual install
      - Option 2: Automated install with Boxstarter
    - Building Node.js
  - Android/Android-based devices (e.g. Firefox OS)
- `Intl` (ECMA-402) support
  - Build with full ICU support (all locales supported by ICU)
    - Unix/macOS
    - Windows
  - Trimmed: `small-icu` (English only) support
    - Unix/macOS
    - Windows
  - Building without Intl support
    - Unix/macOS
    - Windows
  - Use existing installed ICU (Unix/macOS only)

  - Build with a specific ICU
    - Unix/macOS
    - Windows
- Building Node.js with FIPS-compliant OpenSSL
- Building Node.js with external core modules
  - Unix/macOS
  - Windows
- Note for downstream distributors of Node.js

Supported platforms

This list of supported platforms is current as of the branch/release to which it belongs.

Input

Node.js relies on V8 and libuv. We adopt a subset of their supported platforms.

Strategy

There are three support tiers:

- **Tier 1**: These platforms represent the majority of Node.js users. The Node.js Build Working Group maintains infrastructure for full test coverage. Test failures on tier 1 platforms will block releases.
- **Tier 2**: These platforms represent smaller segments of the Node.js user base. The Node.js Build Working Group maintains infrastructure for full test coverage. Test failures on tier 2 platforms will block releases. Infrastructure issues may delay the release of binaries for these platforms.
- **Experimental**: May not compile or test suite may not pass. The core team does not create releases for these platforms. Test failures on experimental platforms do not block releases. Contributions to improve support for these platforms are welcome.

Platforms may move between tiers between major release lines. The table below will reflect those changes.

Platform list

Node.js compilation/execution support depends on operating system, architecture, and libc version. The table below lists the support tier for each supported combination. A list of supported compile toolchains is also supplied for tier 1 platforms.

For production applications, run Node.js on supported platforms only.

Node.js does not support a platform version if a vendor has expired support for it. In other words, Node.js does not support running on End-of-Life (EoL) platforms. This is true regardless of entries in the table below.

Operating System	Architectures	Versions	Support Type	Notes
GNU/Linux	x64	kernel >= 3.10, glibc >= 2.17	Tier 1	e.g. Ubuntu 16.04[^1], Debian 9, EL 7[^2]
GNU/Linux	x64	kernel >= 3.10, musl >= 1.1.19	Experimental	e.g. Alpine 3.8
GNU/Linux	x86	kernel >= 3.10, glibc >= 2.17	Experimental	Downgraded as of Node.js 10

GNU/Linux	arm64	kernel >= 4.5, glibc >= 2.17	Tier 1	e.g. Ubuntu 16.04, Debian 9, EL 7[^3]
GNU/Linux	armv7	kernel >= 4.14, glibc >= 2.24	Tier 1	e.g. Ubuntu 18.04, Debian 9
GNU/Linux	armv6	kernel >= 4.14, glibc >= 2.24	Experimental	Downgraded as of Node.js 12
GNU/Linux	ppc64le >=power8	kernel >= 3.10.0, glibc >= 2.17	Tier 2	e.g. Ubuntu 16.04[^1], EL 7[^2]
GNU/Linux	s390x	kernel >= 3.10.0, glibc >= 2.17	Tier 2	e.g. EL 7[^2]
Windows	x64, x86 (WoW64)	>= Windows 10/Server 2016	Tier 1	[^4],[^5]
Windows	x86 (native)	>= Windows 10/Server 2016	Tier 1 (running) / Experimental (compiling) [^6]	
Windows	x64, x86	Windows 8.1/Server 2012	Experimental	
Windows	arm64	>= Windows 10	Tier 2 (compiling) / Experimental (running)	
macOS	x64	>= 10.15	Tier 1	For notes about compilation see [^7]
macOS	arm64	>= 11	Tier 1	
SmartOS	x64	>= 18	Tier 2	
AIX	ppc64be >=power8	>= 7.2 TL04	Tier 2	
FreeBSD	x64	>= 12.2	Experimental	

[^1]: GCC 8 is not provided on the base platform. Users will need the Toolchain test builds PPA or similar to source a newer compiler.

[^2]: GCC 8 is not provided on the base platform. Users will need the devtoolset-8 or later to source a newer compiler.

[^3]: Older kernel versions may work for ARM64. However the Node.js test infrastructure only tests >= 4.5.

[^4]: On Windows, running Node.js in Windows terminal emulators like `mintty` requires the usage of winpty for the tty channels to work (e.g. `winpty node.exe script.js`). In "Git bash" if you call the node shell alias (`node` without the `.exe` extension), `winpty` is used automatically.

[^5]: The Windows Subsystem for Linux (WSL) is not supported, but the GNU/Linux build process and binaries should work. The community will only address issues that reproduce on native GNU/Linux systems. Issues that only reproduce on WSL should be reported in the WSL issue tracker. Running the Windows binary (`node.exe`) in WSL will not work without workarounds such as stdio redirection.

[^6]: Running Node.js on x86 Windows should work and binaries are provided. However, tests in our infrastructure only run on WoW64. Furthermore, compiling on x86 Windows is Experimental and may not be possible.

[^7]: Our macOS x64 Binaries are compiled with 10.15 as a target. Xcode11 is required to compile.

Supported toolchains

Depending on the host platform, the selection of toolchains may vary.

Operating System	Compiler Versions
Linux	GCC >= 8.3
Windows	Visual Studio >= 2019 with the Windows 10 SDK on a 64-bit host
macOS	Xcode >= 11 (Apple LLVM >= 11)

Official binary platforms and toolchains

Binaries at https://nodejs.org/download/release/ are produced on:

Binary package	Platform and Toolchain
aix-ppc64	AIX 7.2 TL04 on PPC64BE with GCC 8
darwin-x64	macOS 10.15, Xcode Command Line Tools 11 with -mmacosx-version-min=10.15
darwin-arm64 (and .pkg)	macOS 11 (arm64), Xcode Command Line Tools 12 with -mmacosx-version-min=10.15
linux-arm64	CentOS 7 with devtoolset-8 / GCC 8[^8]
linux-armv7l	Cross-compiled on Ubuntu 18.04 x64 with custom GCC toolchain
linux-ppc64le	CentOS 7 with devtoolset-8 / GCC 8[^8]
linux-s390x	RHEL 7 with devtoolset-8 / GCC 8[^8]
linux-x64	CentOS 7 with devtoolset-8 / GCC 8[^8]
win-x64 and win-x86	Windows 2012 R2 (x64) with Visual Studio 2019

[^8]: The Enterprise Linux devtoolset-8 allows us to compile binaries with GCC 8 but linked to the glibc and libstdc++ versions of the host platforms (CentOS 7 / RHEL 7). Therefore, binaries produced on these systems are compatible with glibc >= 2.17 and libstdc++ >= 6.0.20 (`GLIBCXX_3.4.20`). These are available on distributions natively supporting GCC 4.9, such as Ubuntu 14.04 and Debian 8.

OpenSSL asm support

OpenSSL-1.1.1 requires the following assembler version for use of asm support on x86_64 and ia32.

For use of AVX-512,

- gas (GNU assembler) version 2.26 or higher
- nasm version 2.11.8 or higher in Windows

AVX-512 is disabled for Skylake-X by OpenSSL-1.1.1.

For use of AVX2,

- gas (GNU assembler) version 2.23 or higher
- Xcode version 5.0 or higher
- llvm version 3.3 or higher
- nasm version 2.10 or higher in Windows

Please refer to https://www.openssl.org/docs/man1.1.1/man3/OPENSSL_ia32cap.html for details.

If compiling without one of the above, use `configure` with the `--openssl-no-asm` flag. Otherwise, `configure` will fail.

Previous versions of this document

Supported platforms and toolchains change with each major version of Node.js. This document is only valid for the current major version of Node.js. Consult previous versions of this document for older versions of Node.js:

- Node.js 17
- Node.js 16
- Node.js 14
- Node.js 12

Building Node.js on supported platforms

Note about Python

The Node.js project supports Python >= 3 for building and testing.

Unix and macOS

Unix prerequisites

- `gcc` and `g++` >= 8.3 or newer
- GNU Make 3.81 or newer
- Python 3.6, 3.7, 3.8, 3.9, or 3.10 (see note above)
  - For test coverage, your Python installation must include pip.

Installation via Linux package manager can be achieved with:

- Ubuntu, Debian: `sudo apt-get install python3 g++ make python3-pip`
- Fedora: `sudo dnf install python3 gcc-c++ make python3-pip`
- CentOS and RHEL: `sudo yum install python3 gcc-c++ make python3-pip`
- OpenSUSE: `sudo zypper install python3 gcc-c++ make python3-pip`
- Arch Linux, Manjaro: `sudo pacman -S python gcc make python-pip`

FreeBSD and OpenBSD users may also need to install `libexecinfo`.

macOS prerequisites

- Xcode Command Line Tools >= 11 for macOS
- Python 3.6, 3.7, 3.8, 3.9, or 3.10 (see note above)
  - For test coverage, your Python installation must include pip.

macOS users can install the `Xcode Command Line Tools` by running `xcode-select --install`. Alternatively, if you already have the full Xcode installed, you can find them under the menu `Xcode -> Open Developer Tool -> More Developer Tools...`. This step will install `clang`, `clang++`, and `make`.

Building Node.js

If the path to your build directory contains a space, the build will likely fail.

To build Node.js:

```
$ ./configure
$ make -j4
```

We can speed up the builds by using Ninja. For more information, see Building Node.js with Ninja.

The `-j4` option will cause `make` to run 4 simultaneous compilation jobs which may reduce build time. For more information, see the GNU Make Documentation.

The above requires that `python` resolves to a supported version of Python. See Prerequisites.

After building, setting up firewall rules can avoid popups asking to accept incoming network connections when running tests.

Running the following script on macOS will add the firewall rules for the executable `node` in the `out` directory and the symbolic `node` link in the project's root directory.

```
$ sudo ./tools/macos-firewall.sh
```

Installing Node.js

To install this version of Node.js into a system directory:

```
[sudo] make install
```

Running tests

To verify the build:

```
$ make test-only
```

At this point, you are ready to make code changes and re-run the tests.

If you are running tests before submitting a pull request, use:

```
$ make -j4 test
```

`make -j4 test` does a full check on the codebase, including running linters and documentation tests.

To run the linter without running tests, use `make lint` / `vcbuild lint`. It will lint JavaScript, C++, and Markdown files.

If you are updating tests and want to run tests in a single test file (e.g. `test/parallel/test-stream2-transform.js`):

```
$ tools/test.py test/parallel/test-stream2-transform.js
```

You can execute the entire suite of tests for a given subsystem by providing the name of a subsystem:

```
$ tools/test.py -J child-process
```

You can also execute the tests in a tests directory (such as `test/message`):

```
$ tools/test.py -J test/message
```

If you want to check the other options, please refer to the help by using the `--help` option:

```
$ tools/test.py --help
```

You can usually run tests directly with node:

```
$ ./node ./test/parallel/test-stream2-transform.js
```

Remember to recompile with `make -j4` in between test runs if you change code in the `lib` or `src` directories.

The tests attempt to detect support for IPv6 and exclude IPv6 tests if appropriate. If your main interface has IPv6 addresses, then your loopback interface must also have '::1' enabled. For some default installations on Ubuntu, that does not seem to be the case. To enable '::1' on the loopback interface on Ubuntu:

```
sudo sysctl -w net.ipv6.conf.lo.disable_ipv6=0
```

You can use node-code-ide-configs to run/debug tests if your IDE configs are present.

Running coverage

It's good practice to ensure any code you add or change is covered by tests. You can do so by running the test suite with coverage enabled:

```
$ ./configure --coverage
$ make coverage
```

A detailed coverage report will be written to `coverage/index.html` for JavaScript coverage and to `coverage/cxxcoverage.html` for C++ coverage.

If you only want to run the JavaScript tests then you do not need to run the first command (`./configure --coverage`). Run `make coverage-run-js` , to execute JavaScript tests independently of the C++ test suite:

```
$ make coverage-run-js
```

If you are updating tests and want to collect coverage for a single test file (e.g. `test/parallel/test-stream2-transform.js`):

```
$ make coverage-clean
$ NODE_V8_COVERAGE=coverage/tmp tools/test.py test/parallel/test-stream2-
```

```
transform.js
$ make coverage-report-js
```

You can collect coverage for the entire suite of tests for a given subsystem by providing the name of a subsystem:

```
$ make coverage-clean
$ NODE_V8_COVERAGE=coverage/tmp tools/test.py --mode=release child-process
$ make coverage-report-js
```

The `make coverage` command downloads some tools to the project root directory. To clean up after generating the coverage reports:

```
$ make coverage-clean
```

Building the documentation

To build the documentation:

This will build Node.js first (if necessary) and then use it to build the docs:

```
make doc
```

If you have an existing Node.js build, you can build just the docs with:

```
NODE=/path/to/node make doc-only
```

To read the man page:

```
man doc/node.1
```

If you prefer to read the full documentation in a browser, run the following.

```
make docserve
```

This will spin up a static file server and provide a URL to where you may browse the documentation locally.

If you're comfortable viewing the documentation using the program your operating system has associated with the default web browser, run the following.

```
make docopen
```

This will open a file URL to a one-page version of all the browsable HTML documents using the default browser.

To test if Node.js was built correctly:

```
./node -e "console.log('Hello from Node.js ' + process.version)"
```

Building a debug build

If you run into an issue where the information provided by the JS stack trace is not enough, or if you suspect the error happens outside of the JS VM, you can try to build a debug enabled binary:

```
$ ./configure --debug
$ make -j4
```

`make` with `./configure --debug` generates two binaries, the regular release one in `out/Release/node` and a debug binary in `out/Debug/node`, only the release version is actually installed when you run `make install`.

To use the debug build with all the normal dependencies overwrite the release version in the install directory:

```
$ make install PREFIX=/opt/node-debug/
$ cp -a -f out/Debug/node /opt/node-debug/node
```

When using the debug binary, core dumps will be generated in case of crashes. These core dumps are useful for debugging when provided with the corresponding original debug binary and system information.

Reading the core dump requires `gdb` built on the same platform the core dump was captured on (i.e. 64-bit `gdb` for `node` built on a 64-bit system, Linux `gdb` for `node` built on Linux) otherwise you will get errors like `not in executable format: File format not recognized`.

Example of generating a backtrace from the core dump:

```
$ gdb /opt/node-debug/node core.node.8.1535359906
$ backtrace
```

Building an ASAN build

ASAN can help detect various memory related bugs. ASAN builds are currently only supported on linux. If you want to check it on Windows or macOS or you want a consistent toolchain on Linux, you can try Docker (using an image like `gengjiawen/node-build:2020-02-14`).

The `--debug` is not necessary and will slow down build and testing, but it can show clear stacktrace if ASAN hits an issue.

```
$  ./configure --debug --enable-asan && make -j4
$ make test-only
```

Speeding up frequent rebuilds when developing

If you plan to frequently rebuild Node.js, especially if using several branches, installing `ccache` can help to greatly reduce build times. Set up with:

On GNU/Linux:

```
sudo apt install ccache   # for Debian/Ubuntu, included in most Linux distros
export CC="ccache gcc"    # add to your .profile
export CXX="ccache g++"   # add to your .profile
```

On macOS:

```
brew install ccache       # see https://brew.sh
export CC="ccache cc"     # add to ~/.zshrc or other shell config file
export CXX="ccache c++"   # add to ~/.zshrc or other shell config file
```

This will allow for near-instantaneous rebuilds even when switching branches.

When modifying only the JS layer in `lib`, it is possible to externally load it without modifying the executable:

```
$ ./configure --node-builtin-modules-path $(pwd)
```

The resulting binary won't include any JS files and will try to load them from the specified directory. The JS debugger of Visual Studio Code supports this configuration since the November 2020 version and allows for setting breakpoints.

Troubleshooting Unix and macOS builds

Stale builds can sometimes result in `file not found` errors while building. This and some other problems can be resolved with `make distclean`. The `distclean` recipe aggressively removes build artifacts. You will need to build again (`make -j4`). Since all build artifacts have been removed, this rebuild may take a lot more time than previous builds. Additionally, `distclean` removes the file that stores the results of `./configure`. If you ran `./configure` with non-default options (such as `--debug`), you will need to run it again before invoking `make -j4`.

Windows

Prerequisites

Option 1: Manual install

- Python 3.10
- The "Desktop development with C++" workload from Visual Studio 2019 or the "C++ build tools" workload from the Build Tools, with the default optional components
- Basic Unix tools required for some tests, Git for Windows includes Git Bash and tools which can be included in the global `PATH`.
- The NetWide Assembler, for OpenSSL assembler modules. If not installed in the default location, it needs to be manually added to `PATH`. A build with the `openssl-no-asm` option does not need this, nor does a build targeting ARM64 Windows.

Optional requirements to build the MSI installer package:

- The WiX Toolset v3.11 and the Wix Toolset Visual Studio 2019 Extension
- The WiX Toolset v3.14 if building for Windows 10 on ARM (ARM64)

Optional requirements for compiling for Windows 10 on ARM (ARM64):

- Visual Studio 15.9.0 or newer
- Visual Studio optional components
  - Visual C++ compilers and libraries for ARM64
  - Visual C++ ATL for ARM64
- Windows 10 SDK 10.0.17763.0 or newer

Option 2: Automated install with Boxstarter

A Boxstarter script can be used for easy setup of Windows systems with all the required prerequisites for Node.js development. This script will install the following Chocolatey packages:

- Git for Windows with the `git` and Unix tools added to the `PATH`
- Python 3.x
- Visual Studio 2019 Build Tools with Visual C++ workload
- NetWide Assembler

To install Node.js prerequisites using Boxstarter WebLauncher, open https://boxstarter.org/package/nr/url?https://raw.githubusercontent.com/nodejs/node/HEAD/tools/bootstrap/windows_boxstarter with Internet Explorer or Edge browser on the target machine.

Alternatively, you can use PowerShell. Run those commands from an elevated PowerShell terminal:

```
Set-ExecutionPolicy Unrestricted -Force
iex ((New-Object
System.Net.WebClient).DownloadString('https://boxstarter.org/bootstrapper.ps1'))
get-boxstarter -Force
Install-BoxstarterPackage
https://raw.githubusercontent.com/nodejs/node/HEAD/tools/bootstrap/windows_boxstarter
 -DisableReboots
```

The entire installation using Boxstarter will take up approximately 10 GB of disk space.

Building Node.js

If the path to your build directory contains a space or a non-ASCII character, the build will likely fail.

```
> .\vcbuild
```

To run the tests:

```
> .\vcbuild test
```

To test if Node.js was built correctly:

```
> Release\node -e "console.log('Hello from Node.js', process.version)"
```

Android/Android-based devices (e.g. Firefox OS)

Android is not a supported platform. Patches to improve the Android build are welcome. There is no testing on Android in the current continuous integration environment. The participation of people dedicated and determined to improve Android building, testing, and support is encouraged.

Be sure you have downloaded and extracted Android NDK before in a folder. Then run:

```
$ ./android-configure /path/to/your/android-ndk
$ make
```

`Intl` (ECMA-402) support

Intl support is enabled by default.

Build with full ICU support (all locales supported by ICU)

This is the default option.

Unix/macOS

```
$ ./configure --with-intl=full-icu
```

Windows

```
> .\vcbuild full-icu
```

Trimmed: `small-icu` (English only) support

In this configuration, only English data is included, but the full `Intl` (ECMA-402) APIs. It does not need to download any dependencies to function. You can add full data at runtime.

Unix/macOS

```
$ ./configure --with-intl=small-icu
```

Windows

```
> .\vcbuild small-icu
```

Building without Intl support

The `Intl` object will not be available, nor some other APIs such as `String.normalize`.

Unix/macOS

```
$ ./configure --without-intl
```

Windows

```
> .\vcbuild without-intl
```

Use existing installed ICU (Unix/macOS only)

```
$ pkg-config --modversion icu-i18n && ./configure --with-intl=system-icu
```

If you are cross-compiling, your `pkg-config` must be able to supply a path that works for both your host and target environments.

Build with a specific ICU

You can find other ICU releases at the ICU homepage. Download the file named something like `icu4c-**##.#**-src.tgz` (or `.zip`).

To check the minimum recommended ICU, run `./configure --help` and see the help for the `--with-icu-source` option. A warning will be printed during configuration if the ICU version is too old.

Unix/macOS

From an already-unpacked ICU:

```
$ ./configure --with-intl=[small-icu,full-icu] --with-icu-source=/path/to/icu
```

From a local ICU tarball:

```
$ ./configure --with-intl=[small-icu,full-icu] --with-icu-source=/path/to/icu.tgz
```

From a tarball URL:

```
$ ./configure --with-intl=full-icu --with-icu-source=http://url/to/icu.tgz
```

Windows

First unpack latest ICU to `deps/icu` icu4c-##.#-src.tgz (or `.zip`) as `deps/icu` (You'll have: `deps/icu/source/...`)

```
> .\vcbuild full-icu
```

Building Node.js with FIPS-compliant OpenSSL

The current version of Node.js supports FIPS when statically and dynamically linking with OpenSSL 3.0.0 by using the configuration flag `--openssl-is-fips`.

FIPS support when statically linking OpenSSL

FIPS can be supported by specifying the configuration flag `--openssl-is-fips`:

```
$ ./configure --openssl-is-fips
$ make -j8
```

The above command will build and install the FIPS module into the out directory. This includes building fips.so, running the `installfips` command that generates the FIPS configuration file (fipsmodule.cnf), copying and updating openssl.cnf to include the correct path to fipsmodule.cnf and finally uncomment the fips section.

We can then run node specifying `--enable-fips`:

```
$ ./node --enable-fips  -p 'crypto.getFips()'
1
```

The above will use the Node.js default locations for OpenSSL 3.0:

```
$ ./out/Release/openssl-cli version -m -d
OPENSSLDIR: "/nodejs/openssl/out/Release/obj.target/deps/openssl"
MODULESDIR: "/nodejs/openssl/out/Release/obj.target/deps/openssl/lib/openssl-
modules"
```

The OpenSSL configuration files will be found in `OPENSSLDIR` directory above:

```
$ ls -w 1 out/Release/obj.target/deps/openssl/*.cnf
out/Release/obj.target/deps/openssl/fipsmodule.cnf
out/Release/obj.target/deps/openssl/openssl.cnf
```

And the FIPS module will be located in the `MODULESDIR` directory:

```
$ ls out/Release/obj.target/deps/openssl/lib/openssl-modules/
fips.so
```

FIPS support when dynamically linking OpenSSL

For quictls/openssl 3.0 it is possible to enable FIPS when dynamically linking. If you want to build Node.js using openssl-3.0.0+quic, you can follow these steps:

clone OpenSSL source and prepare build

```
git clone git@github.com:quictls/openssl.git

cd openssl

./config \
  --prefix=/path/to/install/dir/ \
  shared \
  enable-fips \
  linux-x86_64
```

The `/path/to/install/dir` is the path in which the `make install` instructions will publish the OpenSSL libraries and such. We will also use this path (and sub-paths) later when compiling Node.js.

compile and install OpenSSL

```
make -j8
make install
make install_ssldirs
make install_fips
```

After the OpenSSL (including FIPS) modules have been compiled and installed (into the `/path/to/install/dir`) by the above instructions we also need to update the OpenSSL configuration file located under `/path/to/install/dir/ssl/openssl.cnf`. Right next to this file, you should find the `fipsmodule.cnf` file - let's add the following to the end of the `openssl.cnf` file.

alter openssl.cnf

```
.include /absolute/path/to/fipsmodule.cnf

# List of providers to load
[provider_sect]
default = default_sect
# The fips section name should match the section name inside the
# included /path/to/install/dir/ssl/fipsmodule.cnf.
fips = fips_sect

[default_sect]
activate = 1
```

You can e.g. accomplish this by running the following command - be sure to replace `/path/to/install/dir/` with the path you have selected. Please make sure that you specify an absolute path for the `.include fipsmodule.cnf` line - using relative paths did not work on my system!

alter openssl.cnf using a script

```
cat <<EOT >> /path/to/install/dir/ssl/openssl.cnf
.include /path/to/install/dir/ssl/fipsmodule.cnf

# List of providers to load
[provider_sect]
default = default_sect
# The fips section name should match the section name inside the
# included /path/to/install/dir/ssl/fipsmodule.cnf.
fips = fips_sect

[default_sect]
activate = 1
EOT
```

As you might have picked a non-custom path for your OpenSSL install dir, we have to export the following two environment variables in order for Node.js to find our OpenSSL modules we built beforehand:

```
export OPENSSL_CONF=/path/to/install/dir/ssl/openssl.cnf
export OPENSSL_MODULES=/path/to/install/dir/lib/ossl-modules
```

build Node.js

```
./configure \
  --shared-openssl \
  --shared-openssl-libpath=/path/to/install/dir/lib \
  --shared-openssl-includes=/path/to/install/dir/include \
  --shared-openssl-libname=crypto,ssl \
  --openssl-is-fips

export LD_LIBRARY_PATH=/path/to/install/dir/lib
```

```
make -j8
```

verify the produced executable

```
ldd ./node
    linux-vdso.so.1 (0x00007ffd7917b000)
    libcrypto.so.81.3 => /path/to/install/dir/lib/libcrypto.so.81.3
(0x00007fd911321000)
    libssl.so.81.3 => /path/to/install/dir/lib/libssl.so.81.3 (0x00007fd91125e000)
    libdl.so.2 => /usr/lib64/libdl.so.2 (0x00007fd911232000)
    libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007fd911039000)
    libm.so.6 => /usr/lib64/libm.so.6 (0x00007fd910ef3000)
    libgcc_s.so.1 => /usr/lib64/libgcc_s.so.1 (0x00007fd910ed9000)
    libpthread.so.0 => /usr/lib64/libpthread.so.0 (0x00007fd910eb5000)
    libc.so.6 => /usr/lib64/libc.so.6 (0x00007fd910cec000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fd9117f2000)
```

If the `ldd` command says that `libcrypto` cannot be found one needs to set `LD_LIBRARY_PATH` to point to the directory used above for `--shared-openssl-libpath` (see previous step).

verify the OpenSSL version

```
./node -p process.versions.openssl
3.0.0-alpha16+quic
```

verify that FIPS is available

```
./node -p 'process.config.variables.openssl_is_fips'
true

./node --enable-fips -p 'crypto.getFips()'
1
```

FIPS support can then be enable via the OpenSSL configuration file or using `--enable-fips` or `--force-fips` command line options to the Node.js executable. See sections Enabling FIPS using Node.js options and Enabling FIPS using OpenSSL config below.

Enabling FIPS using Node.js options

This is done using one of the Node.js options `--enable-fips` or `--force-fips`, for example:

```
$ node --enable-fips -p 'crypto.getFips()'
```

Enabling FIPS using OpenSSL config

This example show that using OpenSSL's configuration file, FIPS can be enabled without specifying the `--enable-fips` or `--force-fips` options by setting `default_properties = fips=yes` in the FIPS configuration file. See link for details.

For this to work the OpenSSL configuration file (default openssl.cnf) needs to be updated. The following shows an example:

```
openssl_conf = openssl_init

.include /path/to/install/dir/ssl/fipsmodule.cnf

[openssl_init]
providers = prov
alg_section = algorithm_sect

[prov]
fips = fips_sect
default = default_sect

[default_sect]
activate = 1

[algorithm_sect]
default_properties = fips=yes
```

After this change Node.js can be run without the `--enable-fips` or `--force-fips` options.

Building Node.js with external core modules

It is possible to specify one or more JavaScript text files to be bundled in the binary as built-in modules when building Node.js.

Unix/macOS

This command will make `/root/myModule.js` available via `require('/root/myModule')` and `./myModule2.js` available via `require('myModule2')`.

```
$ ./configure --link-module '/root/myModule.js' --link-module './myModule2.js'
```

Windows

To make `./myModule.js` available via `require('myModule')` and `./myModule2.js` available via `require('myModule2')`:

```
> .\vcbuild link-module './myModule.js' link-module './myModule2.js'
```

Note for downstream distributors of Node.js

The Node.js ecosystem is reliant on ABI compatibility within a major release. To maintain ABI compatibility it is required that distributed builds of Node.js be built against the same version of dependencies, or similar versions that do not break their ABI compatibility, as those released by Node.js for any given `NODE_MODULE_VERSION` (located in `src/node_version.h`).

When Node.js is built (with an intention to distribute) with an ABI incompatible with the official Node.js builds (e.g. using a ABI incompatible version of a dependency), please reserve and use a custom `NODE_MODULE_VERSION` by opening a pull request against the registry available at https://github.com/nodejs/node/blob/HEAD/doc/abi_version_registry.json.