

ProTeamsPricingDocumentation

npm

Sign UpSign In

Search packages

Search

bcrypt

DT

5.1.0 • Public • Published 6 months ago

Readme

CodeBeta

2 Dependencies

3,824 Dependents

53 Versions

# node.bcrypt.js

ci

passing

BUILD PASSING

A library to help you hash passwords.

You can read about [bcrypt in Wikipedia](#) as well as in the following article: [How To Safely Store A Password](#)

## If You Are Submitting Bugs or Issues

Please verify that the NodeJS version you are using is a *stable* version; Unstable versions are currently not supported and issues created while using an unstable version will be closed.

If you are on a stable version of NodeJS, please provide a sufficient code snippet or log files for installation issues. The code snippet does not require you to include confidential information. However, it must provide enough information so the problem can be replicable, or it may be closed without an explanation.

## Version Compatibility

*Please upgrade to atleast v5.0.0 to avoid security issues mentioned below.*

Node Version	Bcrypt Version
0.4	<= 0.4

Node Version	Bcrypt Version
0.6, 0.8, 0.10	<code>&gt;= 0.5</code>
0.11	<code>&gt;= 0.8</code>
4	<code>&lt;= 2.1.0</code>
8	<code>&gt;= 1.0.3 &lt; 4.0.0</code>
10, 11	<code>&gt;= 3</code>
12 onwards	<code>&gt;= 3.0.6</code>

`node-gyp` only works with stable/released versions of node. Since the `bcrypt` module uses `node-gyp` to build and install, you'll need a stable version of node to use bcrypt. If you do not, you'll likely see an error that starts with:

```
gyp ERR! stack Error: "pre" versions of node cannot be installed, use the --nodedir flag instead.
```

## Security Issues And Concerns

Per bcrypt implementation, only the first 72 bytes of a string are used. Any extra bytes are ignored when matching passwords. Note that this is not the first *72 characters*. It is possible for a string to contain less than 72 characters, while taking up more than 72 bytes (e.g. a UTF-8 encoded string containing emojis).

As should be the case with any security tool, anyone using this library should scrutinise it. If you find or suspect an issue with the code, please bring it to the maintainers' attention. We will spend some time ensuring that this library is as secure as possible.

Here is a list of BCrypt-related security issues/concerns that have come up over the years.

- An [issue with passwords](#) was found with a version of the Blowfish algorithm developed for John the Ripper. This is not present in the OpenBSD version and is thus not a problem for this module. HT [zooko](#).
- Versions `< 5.0.0` suffer from bcrypt wrap-around bug and *will truncate passwords `>= 255 characters` leading to severely weakened passwords*. Please upgrade at earliest. See [this wiki page](#) for more details.
- Versions `< 5.0.0` *do not handle NUL characters inside passwords properly leading to all subsequent characters being dropped and thus resulting in severely weakened passwords*. Please upgrade at earliest. See [this wiki page](#) for more details.

## Compatibility Note

This library supports `$2a$` and `$2b$` prefix bcrypt hashes. `$2x$` and `$2y$` hashes are specific to bcrypt implementation developed for John the Ripper. In theory, they should be

compatible with `$2b$` prefix.

Compatibility with hashes generated by other languages is not 100% guaranteed due to difference in character encodings. However, it should not be an issue for most cases.

#### Migrating from v1.0.x

Hashes generated in earlier version of `bcrypt` remain 100% supported in `v2.x.x` and later versions. In most cases, the migration should be a bump in the `package.json`.

Hashes generated in `v2.x.x` using the defaults parameters will not work in earlier versions.

## Dependencies

---

- NodeJS
- `node-gyp`
- Please check the dependencies for this tool at: <https://github.com/nodejs/node-gyp>
- Windows users will need the options for c# and c++ installed with their visual studio instance.
- Python 2.x/3.x
- OpenSSL - This is only required to build the `bcrypt` project if you are using versions  $\leq 0.7.7$ . Otherwise, we're using the builtin node crypto bindings for seed data (which use the same OpenSSL code paths we were, but don't have the external dependency).

## Install via NPM

---

```
npm install bcrypt
```

**Note:** OS X users using Xcode 4.3.1 or above may need to run the following command in their terminal prior to installing if errors occur regarding `xcodebuild`: `sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer`

*Pre-built binaries for various NodeJS versions are made available on a best-effort basis.*

Only the current stable and supported LTS releases are actively tested against.

*There may be an interval between the release of the module and the availability of the compiled modules.*

Currently, we have pre-built binaries that support the following platforms:

1. Windows x32 and x64
2. Linux x64 (Glibc and musl)
3. macOS

If you face an error like this:

node-pre-gyp ERR! Tried to download(404): https://github.com/kelektiv/node.bcrypt.js/releases/

make sure you have the appropriate dependencies installed and configured for your platform. You can find installation instructions for the dependencies for some common platforms [in this page](#).

## Usage

---

async (recommended)

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const myPlaintextPassword = 's0/\\/\P4$$w0rd';
const someOtherPlaintextPassword = 'not_bacon';
```

To hash a password:

---

Technique 1 (generate a salt and hash on separate function calls):

```
bcrypt.genSalt(saltRounds, function(err, salt) {
  bcrypt.hash(myPlaintextPassword, salt, function(err, hash) {
    // Store hash in your password DB.
  });
});
```

Technique 2 (auto-gen a salt and hash):

```
bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) {
  // Store hash in your password DB.
});
```

Note that both techniques achieve the same end-result.

To check a password:

---

```
// Load hash from your password DB.
bcrypt.compare(myPlaintextPassword, hash, function(err, result) {
  // result == true
});
bcrypt.compare(someOtherPlaintextPassword, hash, function(err, result) {
  // result == false
});
```

## A Note on Timing Attacks

### with promises

bcrypt uses whatever `Promise` implementation is available in `global.Promise`.

NodeJS  $\geq 0.12$  has a native `Promise` implementation built in. However, this should work in any Promises/A+ compliant implementation.

Async methods that accept a callback, return a `Promise` when callback is not specified if `Promise` support is available.

```
bcrypt.hash(myPlaintextPassword, saltRounds).then(function(hash) {  
  // Store hash in your password DB.  
});  
  
// Load hash from your password DB.  
bcrypt.compare(myPlaintextPassword, hash).then(function(result) {  
  // result == true  
});  
bcrypt.compare(someOtherPlaintextPassword, hash).then(function(result)  
  // result == false  
});
```

This is also compatible with `async/await`

```
async function checkUser(username, password) {  
  //... fetch user from a db etc.  
  
  const match = await bcrypt.compare(password, user.passwordHash);  
  
  if(match) {  
    //login  
  }  
  
  //...  
}
```

### ESM import

```
import bcrypt from "bcrypt";  
  
// later  
await bcrypt.compare(password, hash);
```

### sync

```
const bcrypt = require('bcrypt');
```

```
const saltRounds = 10;
const myPlaintextPassword = 's0/\\/\P4$$w0rD';
const someOtherPlaintextPassword = 'not_bacon';
```

To hash a password:

---

Technique 1 (generate a salt and hash on separate function calls):

```
const salt = bcrypt.genSaltSync(saltRounds);
const hash = bcrypt.hashSync(myPlaintextPassword, salt);
// Store hash in your password DB.
```

Technique 2 (auto-gen a salt and hash):

```
const hash = bcrypt.hashSync(myPlaintextPassword, saltRounds);
// Store hash in your password DB.
```

As with async, both techniques achieve the same end-result.

To check a password:

---

```
// Load hash from your password DB.
bcrypt.compareSync(myPlaintextPassword, hash); // true
bcrypt.compareSync(someOtherPlaintextPassword, hash); // false
```

## A Note on Timing Attacks

Why is async mode recommended over sync mode?

We recommend using async API if you use `bcrypt` on a server. Bcrypt hashing is CPU intensive which will cause the sync APIs to block the event loop and prevent your application from servicing any inbound requests or events. The async version uses a thread pool which does not block the main event loop.

## API

---

`BCrypt`.

- `genSaltSync(rounds, minor)`
  - `rounds` - [OPTIONAL] - the cost of processing the data. (default - 10)
  - `minor` - [OPTIONAL] - minor version of bcrypt to use. (default - b)
- `genSalt(rounds, minor, cb)`
  - `rounds` - [OPTIONAL] - the cost of processing the data. (default - 10)
  - `minor` - [OPTIONAL] - minor version of bcrypt to use. (default - b)
  - `cb` - [OPTIONAL] - a callback to be fired once the salt has been generated. uses `eio` making it asynchronous. If `cb` is not specified, a `Promise` is returned if `Promise` support is available.

- `err` - First parameter to the callback detailing any errors.
- `salt` - Second parameter to the callback providing the generated salt.
- `hashSync(data, salt)`
  - `data` - [REQUIRED] - the data to be encrypted.
  - `salt` - [REQUIRED] - the salt to be used to hash the password. if specified as a number then a salt will be generated with the specified number of rounds and used (see example under **Usage**).
- `hash(data, salt, cb)`
  - `data` - [REQUIRED] - the data to be encrypted.
  - `salt` - [REQUIRED] - the salt to be used to hash the password. if specified as a number then a salt will be generated with the specified number of rounds and used (see example under **Usage**).
  - `cb` - [OPTIONAL] - a callback to be fired once the data has been encrypted. uses `eo` making it asynchronous. If `cb` is not specified, a `Promise` is returned if Promise support is available.
    - `err` - First parameter to the callback detailing any errors.
    - `encrypted` - Second parameter to the callback providing the encrypted form.
- `compareSync(data, encrypted)`
  - `data` - [REQUIRED] - data to compare.
  - `encrypted` - [REQUIRED] - data to be compared to.
- `compare(data, encrypted, cb)`
  - `data` - [REQUIRED] - data to compare.
  - `encrypted` - [REQUIRED] - data to be compared to.
  - `cb` - [OPTIONAL] - a callback to be fired once the data has been compared. uses `eo` making it asynchronous. If `cb` is not specified, a `Promise` is returned if Promise support is available.
    - `err` - First parameter to the callback detailing any errors.
    - `same` - Second parameter to the callback providing whether the data and encrypted forms match [`true` | `false`].
- `getRounds(encrypted)` - return the number of rounds used to encrypt a given hash
  - `encrypted` - [REQUIRED] - hash from which the number of rounds used should be extracted.

## A Note on Rounds

---

A note about the cost: when you are hashing your data, the module will go through a series of rounds to give you a secure hash. The value you submit is not just the number of rounds the module will go through to hash your data. The module will use the value you enter and go through  $2^{\text{rounds}}$  hashing iterations.

From @garthk, on a 2GHz core you can roughly expect:

rounds=8 : ~40 hashes/sec

rounds=9 : ~20 hashes/sec

rounds=10: ~10 hashes/sec

```
rounds=11: ~5 hashes/sec
rounds=12: 2-3 hashes/sec
rounds=13: ~1 sec/hash
rounds=14: ~1.5 sec/hash
rounds=15: ~3 sec/hash
rounds=25: ~1 hour/hash
rounds=31: 2-3 days/hash
```

## A Note on Timing Attacks

---

Because it's come up multiple times in this project and other bcrypt projects, it needs to be said. The `bcrypt` library is not susceptible to timing attacks. From [codahale/bcrypt-ruby#42](#):

One of the desired properties of a cryptographic hash function is preimage attack resistance, which means there is no shortcut for generating a message which, when hashed, produces a specific digest.

A great thread on this, in much more detail can be found @ [codahale/bcrypt-ruby#43](#)

If you're unfamiliar with timing attacks and want to learn more you can find a great writeup @ [A Lesson In Timing Attacks](#)

However, timing attacks are real. And the comparison function is *not* time safe. That means that it may exit the function early in the comparison process. Timing attacks happen because of the above. We don't need to be careful that an attacker will learn anything, and our comparison function provides a comparison of hashes. It is a utility to the overall purpose of the library. If you end up using it for something else, we cannot guarantee the security of the comparator. Keep that in mind as you use the library.

## Hash Info

---

The characters that comprise the resultant hash are

```
./ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789$.
```

Resultant hashes will be 60 characters long and they will include the salt among other parameters, as follows:

```
$(algorithm)$(cost)$(salt)[hash]
```

- 2 chars hash algorithm identifier prefix. "\$2a\$" or "\$2b\$" indicates BCrypt
- Cost-factor (n). Represents the exponent used to determine how many iterations  $2^n$
- 16-byte (128-bit) salt, base64 encoded to 22 characters
- 24-byte (192-bit) hash, base64 encoded to 31 characters



### Example:

```
$2b$10$n0UIs5kJ7naTuTFkBy1veuK0kSxUFxfuaOKdOKf9xYT0KKIGSJwFa
| | |
| | | hash-value = K0kSxUFxfuaOKdOKf9xYT0KKIGSJwFa
| | |
| | salt = n0UIs5kJ7naTuTFkBy1veu
| |
| cost-factor => 10 = 2^10 rounds
|
hash-algorithm identifier => 2b = BCrypt
```

## Testing

If you create a pull request, tests better pass :)

```
npm install
npm test
```

## Credits

The code for this comes from a few sources:

- [blowfish.cc](#) - OpenBSD
- [bcrypt.cc](#) - OpenBSD
- [bcrypt::gen\\_salt](#) - [gen\\_salt inclusion to bcrypt](#)
- [bcrypt\\_node.cc](#) - me

## Contributors

- [Antonio Salazar Cardozo](#) - Early MacOS X support (when we used libbsd)
- [Ben Glow](#) - Fixes for thread safety with async calls
- [Van Nguyen](#) - Found a timing attack in the comparator
- [NewITFarmer](#) - Initial Cygwin support
- [David Trejo](#) - packaging fixes
- [Alfred Westerveld](#) - packaging fixes
- [Vincent Côté-Roy](#) - Testing around concurrency issues
- [Lloyd Hilaiei](#) - Documentation fixes
- [Roman Shtylman](#) - Code refactoring, general rot reduction, compile options, better memory management with delete and new, and an upgrade to libuv over eio/ev.
- [Vadim Graboy](#)s - Code changes to support 0.5.5+
- [Ben Noordhuis](#) - Fixed a thread safety issue in nodejs that was perfectly mappable to this module.

- **Nate Rajlich** - Bindings and build process.
- **Sean McArthur** - Windows Support
- **Fanie Oosthuysen** - Windows Support
- **Amitosh Swain Mahapatra** - \$2b\$ hash support, ES6 Promise support
- **Nicola Del Gobbo** - Initial implementation with N-API

License

Unless stated elsewhere, file headers or otherwise, the license as stated in the LICENSE file.

Install

Keywords

npm bcrypt

**bcrypt password auth authentication encryption crypt crypto**

Repository [github.com/kelektiv/node.bcrypt.js](https://github.com/kelektiv/node.bcrypt.js)






Homepage [github.com/kelektiv/node.bcrypt.js#readme](https://github.com/kelektiv/node.bcrypt.js#readme)



Version	License
5.1.0	MIT
Unpacked Size	Total Files
143 kB	27
Issues	Pull Requests
25	3

Last publish  
6 months ago

Collaborators



[Try on RunKit](#)

[Report malware](#)



## Support

[Help](#)

[Advisories](#)

[Status](#)

[Contact npm](#)

## Company

[About](#)

[Blog](#)

[Press](#)

## Terms & Policies

[Policies](#)

[Terms of Use](#)

[Code of Conduct](#)

[Privacy](#)