safer-buffer

Modern Buffer API polyfill without footguns, working on Node.js from 0.8 to current.

How to use?

First, port all `Buffer()` and `new Buffer()` calls to `Buffer.alloc()` and `Buffer.from()` API.

Then, to achieve compatibility with outdated Node.js versions (`<4.5.0` and 5.x `<5.9.0`), use `const Buffer = require('safer-buffer').Buffer` in all files where you make calls to the new Buffer API. *Use* `var` *instead of* `const` *if you need that for your Node.js version range support.*

Also, see the porting Buffer guide.

Do I need it?

Hopefully, not — dropping support for outdated Node.js versions should be fine nowdays, and that is the recommended path forward. You *do* need to port to the `Buffer.alloc()` and `Buffer.from()` though.

See the porting guide for a better description.

Why not safe-buffer?

In short: while `safe-buffer` *serves as a polyfill for the new API, it allows old API usage and itself contains footguns.*

`safe-buffer` could be used safely to get the new API while still keeping support for older Node.js versions (like this module), but while analyzing ecosystem usage of the old Buffer API I found out that `safe-buffer` is itself causing problems in some cases.

For example, consider the following snippet:

```
$ cat example.unsafe.js
console.log(Buffer(20))
$ ./node-v6.13.0-linux-x64/bin/node example.unsafe.js
<Buffer 0a 00 00 00 00 00 00 00 28 13 de 02 00 00 00 00 05 00 00 00>
$ standard example.unsafe.js
standard: Use JavaScript Standard Style (https://standardjs.com)
  /home/chalker/repo/safer-buffer/example.unsafe.js:2:13: 'Buffer()' was deprecated
since v6. Use 'Buffer.alloc()' or 'Buffer.from()' (use
'https://www.npmjs.com/package/safe-buffer' for '<4.5.0') instead.
```

This is allocates and writes to console an uninitialized chunk of memory. standard linter (among others) catch that and warn people to avoid using unsafe API.

Let's now throw in `safe-buffer` !

```
$ cat example.safe-buffer.js
const Buffer = require('safe-buffer').Buffer
console.log(Buffer(20))
$ standard example.safe-buffer.js
$ ./node-v6.13.0-linux-x64/bin/node example.safe-buffer.js
<Buffer 08 00 00 00 00 00 00 00 28 58 01 82 fe 7f 00 00 00 00 00 00>
```

See the problem? Adding in `safe-buffer` *magically removes the lint warning*, but the behavior remains identical to what we had before, and when launched on Node.js 6.x LTS — this dumps out chunks of uninitialized memory. *And this code will still emit runtime warnings on Node.js 10.x and above.*

That was done by design. I first considered changing `safe-buffer`, prohibiting old API usage or emitting warnings on it, but that significantly diverges from `safe-buffer` design. After some discussion, it was decided to move my approach into a separate package, and *this is that separate package*.

This footgun is not imaginary — I observed top-downloaded packages doing that kind of thing, «fixing» the lint warning by blindly including `safe-buffer` without any actual changes.

Also in some cases, even if the API *was* migrated to use of safe Buffer API — a random pull request can bring unsafe Buffer API usage back to the codebase by adding new calls — and that could go unnoticed even if you have a linter prohibiting that (becase of the reason stated above), and even pass CI. *I also observed that being done in popular packages.*

Some examples:

- webdriverio (a module with 548 759 downloads/month),
- websocket-stream (218 288 d/m, fix in maxogden/websocket-stream#142),
- node-serialport (113 138 d/m, fix in node-serialport/node-serialport#1510),
- karma (3 973 193 d/m, fix in karma-runner/karma#2947),
- spdy-transport (5 970 727 d/m, fix in spdy-http2/spdy-transport#53).
- And there are a lot more over the ecosystem.

I filed a PR at mysticatea/eslint-plugin-node#110 to partially fix that (for cases when that lint rule is used), but it is a semver-major change for linter rules and presets, so it would take significant time for that to reach actual setups. *It also hasn't been released yet (2018-03-20).*

Also, `safer-buffer` discourages the usage of `.allocUnsafe()`, which is often done by a mistake. It still supports it with an explicit concern barier, by placing it under `require('safer-buffer/dangereous')`.

But isn't throwing bad?

Not really. It's an error that could be noticed and fixed early, instead of causing havoc later like unguarded `new Buffer()` calls that end up receiving user input can do.

This package affects only the files where `var Buffer = require('safer-buffer').Buffer` was done, so it is really simple to keep track of things and make sure that you don't mix old API usage with that. Also, CI should hint anything that you might have missed.

New commits, if tested, won't land new usage of unsafe Buffer API this way. *Node.js 10.x also deals with that by printing a runtime depecation warning.*

Would it affect third-party modules?

No, unless you explicitly do an awful thing like monkey-patching or overriding the built-in `Buffer`. Don't do that.

But I don't want throwing...

That is also fine!

Also, it could be better in some cases when you don't comprehensive enough test coverage.

In that case — just don't override `Buffer` and use `var SaferBuffer = require('safer-buffer').Buffer` instead.

That way, everything using `Buffer` natively would still work, but there would be two drawbacks:

- `Buffer.from` / `Buffer.alloc` won't be polyfilled — use `SaferBuffer.from` and `SaferBuffer.alloc` instead.
- You are still open to accidentally using the insecure deprecated API — use a linter to catch that.

Note that using a linter to catch accidential `Buffer` constructor usage in this case is strongly recommended. `Buffer` is not overriden in this usecase, so linters won't get confused.

«Without footguns»?

Well, it is still possible to do *some* things with `Buffer` API, e.g. accessing `.buffer` property on older versions and duping things from there. You shouldn't do that in your code, probabably.

The intention is to remove the most significant footguns that affect lots of packages in the ecosystem, and to do it in the proper way.

Also, this package doesn't protect against security issues affecting some Node.js versions, so for usage in your own production code, it is still recommended to update to a Node.js version supported by upstream.