Porting to the Buffer.from/Buffer.alloc API

Overview

- Variant 1: Drop support for Node.js ≤ 4.4.x and 5.0.0 — 5.9.x. (*recommended*)
- Variant 2: Use a polyfill
- Variant 3: manual detection, with safeguards

Finding problematic bits of code using grep

Just run `grep -nrE '[^a-zA-Z](Slow)?Buffer\s*\(' --exclude-dir node_modules`.

It will find all the potentially unsafe places in your own code (with some considerably unlikely exceptions).

Finding problematic bits of code using Node.js 8

If you're using Node.js ≥ 8.0.0 (which is recommended), Node.js exposes multiple options that help with finding the relevant pieces of code:

- `--trace-warnings` will make Node.js show a stack trace for this warning and other warnings that are printed by Node.js.
- `--trace-deprecation` does the same thing, but only for deprecation warnings.
- `--pending-deprecation` will show more types of deprecation warnings. In particular, it will show the `Buffer()` deprecation warning, even on Node.js 8.

You can set these flags using an environment variable:

```
$ export NODE_OPTIONS='--trace-warnings --pending-deprecation'
$ cat example.js
'use strict';
const foo = new Buffer('foo');
$ node example.js
(node:7147) [DEP0005] DeprecationWarning: The Buffer() and new Buffer() constructors
are not recommended for use due to security and usability concerns. Please use the
new Buffer.alloc(), Buffer.allocUnsafe(), or Buffer.from() construction methods
instead.
    at showFlaggedDeprecation (buffer.js:127:13)
    at new Buffer (buffer.js:148:3)
    at Object.<anonymous> (/path/to/example.js:2:13)
    [... more stack trace lines ...]
```

Finding problematic bits of code using linters

Eslint rules no-buffer-constructor or node/no-deprecated-api also find calls to deprecated `Buffer()` API. Those rules are included in some pre-sets.

There is a drawback, though, that it doesn't always work correctly when `Buffer` is overriden e.g. with a polyfill, so recommended is a combination of this and some other method described above.

Variant 1: Drop support for Node.js ≤ 4.4.x and 5.0.0 — 5.9.x.

This is the recommended solution nowadays that would imply only minimal overhead.

The Node.js 5.x release line has been unsupported since July 2016, and the Node.js 4.x release line reaches its End of Life in April 2018 (→ Schedule). This means that these versions of Node.js will *not* receive any updates, even in case of security issues, so using these release lines should be avoided, if at all possible.

What you would do in this case is to convert all `new Buffer()` or `Buffer()` calls to use `Buffer.alloc()` or `Buffer.from()`, in the following way:

- For `new Buffer(number)`, replace it with `Buffer.alloc(number)`.
- For `new Buffer(string)` (or `new Buffer(string, encoding)`), replace it with `Buffer.from(string)` (or `Buffer.from(string, encoding)`).
- For all other combinations of arguments (these are much rarer), also replace `new Buffer(...arguments)` with `Buffer.from(...arguments)`.

Note that `Buffer.alloc()` is also *faster* on the current Node.js versions than `new Buffer(size).fill(0)`, which is what you would otherwise need to ensure zero-filling.

Enabling eslint rule no-buffer-constructor or node/no-deprecated-api is recommended to avoid accidential unsafe Buffer API usage.

There is also a JSCodeshift codemod for automatically migrating Buffer constructors to `Buffer.alloc()` or `Buffer.from()`. Note that it currently only works with cases where the arguments are literals or where the constructor is invoked with two arguments.

If you currently support those older Node.js versions and dropping them would be a semver-major change for you, or if you support older branches of your packages, consider using Variant 2 or Variant 3 on older branches, so people using those older branches will also receive the fix. That way, you will eradicate potential issues caused by unguarded Buffer API usage and your users will not observe a runtime deprecation warning when running your code on Node.js 10.

Variant 2: Use a polyfill

Utilize safer-buffer as a polyfill to support older Node.js versions.

You would take exacly the same steps as in Variant 1, but with a polyfill `const Buffer = require('safer-buffer').Buffer` in all files where you use the new `Buffer` api.

Make sure that you do not use old `new Buffer` API — in any files where the line above is added, using old `new Buffer()` API will *throw*. It will be easy to notice that in CI, though.

Alternatively, you could use buffer-from and/or buffer-alloc ponyfills — those are great, the only downsides being 4 deps in the tree and slightly more code changes to migrate off them (as you would be using e.g. `Buffer.from` under a different name). If you need only `Buffer.from` polyfilled — `buffer-from` alone which comes with no extra dependencies.

Alternatively, you could use safe-buffer — it also provides a polyfill, but takes a different approach which has it's drawbacks. It will allow you to also use the older `new Buffer()` API in your code, though — but that's arguably a benefit, as it is problematic, can cause issues in your code, and will start emitting runtime deprecation warnings starting with Node.js 10.

Note that in either case, it is important that you also remove all calls to the old Buffer API manually — just throwing in `safe-buffer` doesn't fix the problem by itself, it just provides a polyfill for the new API. I have seen people doing that mistake.

Enabling eslint rule no-buffer-constructor or node/no-deprecated-api is recommended.

Don't forget to drop the polyfill usage once you drop support for Node.js < 4.5.0.

Variant 3 — manual detection, with safeguards

This is useful if you create Buffer instances in only a few places (e.g. one), or you have your own wrapper around them.

Buffer(0)

This special case for creating empty buffers can be safely replaced with `Buffer.concat([])`, which returns the same result all the way down to Node.js 0.8.x.

Buffer(notNumber)

Before:

```
var buf = new Buffer(notNumber, encoding);
```

After:

```
var buf;
if (Buffer.from && Buffer.from !== Uint8Array.from) {
  buf = Buffer.from(notNumber, encoding);
} else {
  if (typeof notNumber === 'number')
    throw new Error('The "size" argument must be of type number.');
  buf = new Buffer(notNumber, encoding);
}
```

`encoding` is optional.

Note that the `typeof notNumber` before `new Buffer` is required (for cases when `notNumber` argument is not hard-coded) and *is not caused by the deprecation of Buffer constructor* — it's exactly *why* the Buffer constructor is deprecated. Ecosystem packages lacking this type-check caused numereous security issues — situations when unsanitized user input could end up in the `Buffer(arg)` create problems ranging from DoS to leaking sensitive information to the attacker from the process memory.

When `notNumber` argument is hardcoded (e.g. literal `"abc"` or `[0,1,2]`), the `typeof` check can be omitted.

Also note that using TypeScript does not fix this problem for you — when libs written in `TypeScript` are used from JS, or when user input ends up there — it behaves exactly as pure JS, as all type checks are translation-time only and are not present in the actual JS code which TS compiles to.

Buffer(number)

For Node.js 0.10.x (and below) support:

```
var buf;
if (Buffer.alloc) {
  buf = Buffer.alloc(number);
} else {
```

```
  buf = new Buffer(number);
  buf.fill(0);
}
```

Otherwise (Node.js ≥ 0.12.x):

```
const buf = Buffer.alloc ? Buffer.alloc(number) : new Buffer(number).fill(0);
```

Regarding Buffer.allocUnsafe

Be extra cautious when using `Buffer.allocUnsafe`:

- Don't use it if you don't have a good reason to
  - e.g. you probably won't ever see a performance difference for small buffers, in fact, those might be even faster with `Buffer.alloc()`,
  - if your code is not in the hot code path — you also probably won't notice a difference,
  - keep in mind that zero-filling minimizes the potential risks.
- If you use it, make sure that you never return the buffer in a partially-filled state,
  - if you are writing to it sequentially — always truncate it to the actuall written length

Errors in handling buffers allocated with `Buffer.allocUnsafe` could result in various issues, ranged from undefined behaviour of your code to sensitive data (user input, passwords, certs) leaking to the remote attacker.

Note that the same applies to `new Buffer` usage without zero-filling, depending on the Node.js version (and lacking type checks also adds DoS to the list of potential problems).

FAQ

What is wrong with the `Buffer` constructor?

The `Buffer` constructor could be used to create a buffer in many different ways:

- `new Buffer(42)` creates a `Buffer` of 42 bytes. Before Node.js 8, this buffer contained *arbitrary memory* for performance reasons, which could include anything ranging from program source code to passwords and encryption keys.
- `new Buffer('abc')` creates a `Buffer` that contains the UTF-8-encoded version of the string `'abc'`. A second argument could specify another encoding: For example, `new Buffer(string, 'base64')` could be used to convert a Base64 string into the original sequence of bytes that it represents.
- There are several other combinations of arguments.

This meant that, in code like `var buffer = new Buffer(foo);`, *it is not possible to tell what exactly the contents of the generated buffer are* without knowing the type of `foo`.

Sometimes, the value of `foo` comes from an external source. For example, this function could be exposed as a service on a web server, converting a UTF-8 string into its Base64 form:

```
function stringToBase64(req, res) {
  // The request body should have the format of `{ string: 'foobar' }`
  const rawBytes = new Buffer(req.body.string)
  const encoded = rawBytes.toString('base64')
```

```
  res.end({ encoded: encoded })
}
```

Note that this code does *not* validate the type of `req.body.string`:

- `req.body.string` is expected to be a string. If this is the case, all goes well.
- `req.body.string` is controlled by the client that sends the request.
- If `req.body.string` is the *number* `50`, the `rawBytes` would be 50 bytes:
  - Before Node.js 8, the content would be uninitialized
  - After Node.js 8, the content would be `50` bytes with the value `0`

Because of the missing type check, an attacker could intentionally send a number as part of the request. Using this, they can either:

- Read uninitialized memory. This **will** leak passwords, encryption keys and other kinds of sensitive information. (Information leak)
- Force the program to allocate a large amount of memory. For example, when specifying `500000000` as the input value, each request will allocate 500MB of memory. This can be used to either exhaust the memory available of a program completely and make it crash, or slow it down significantly. (Denial of Service)

Both of these scenarios are considered serious security issues in a real-world web server context.

when using `Buffer.from(req.body.string)` instead, passing a number will always throw an exception instead, giving a controlled behaviour that can always be handled by the program.

The `Buffer()` constructor has been deprecated for a while. Is this really an issue?

Surveys of code in the `npm` ecosystem have shown that the `Buffer()` constructor is still widely used. This includes new code, and overall usage of such code has actually been *increasing*.