# safe-buffer [travis](https://travis-ci.org/feross/safe-buffer) [npm](https://npmjs.org/package/safe-buffer) [downloads](https://npmjs.org/package/safe-buffer) [javascript style guide](https://standardjs.com)

#### Safer Node.js Buffer API

**Use the new Node.js Buffer APIs (Buffer.from, Buffer.alloc, Buffer.allocUnsafe, Buffer.allocUnsafeSlow) in all versions of Node.js.**

**Uses the built-in implementation when available.**

## install

npm install safe-buffer

## usage

The goal of this package is to provide a safe replacement for the node.js Buffer.

It's a drop-in replacement for Buffer. You can use it by adding one require line to the top of your node.js modules:

var Buffer = require('safe-buffer').Buffer

// Existing buffer code will continue to work without issues:

new Buffer('hey', 'utf8')

new Buffer([1, 2, 3], 'utf8')

new Buffer(obj)

new Buffer(16) // create an uninitialized buffer (potentially unsafe)

// But you can use these new explicit APIs to make clear what you want:

Buffer.from('hey', 'utf8') // convert from many types to a Buffer

Buffer.alloc(16) // create a zero-filled buffer (safe)

Buffer.allocUnsafe(16) // create an uninitialized buffer (potentially unsafe)

## api

### Class Method: Buffer.from(array)

* array {Array}

Allocates a new Buffer using an array of octets.

const buf = Buffer.from([0x62,0x75,0x66,0x66,0x65,0x72]);

// creates a new Buffer containing ASCII bytes

// ['b','u','f','f','e','r']

A TypeError will be thrown if array is not an Array.

### Class Method: Buffer.from(arrayBuffer[, byteOffset[, length]])

* arrayBuffer {ArrayBuffer} The .buffer property of a TypedArray or a new ArrayBuffer()
* byteOffset {Number} Default: 0
* length {Number} Default: arrayBuffer.length - byteOffset

When passed a reference to the .buffer property of a TypedArray instance, the newly created Buffer will share the same allocated memory as the TypedArray.

const arr = new Uint16Array(2);

arr[0] = 5000;

arr[1] = 4000;

const buf = Buffer.from(arr.buffer); // shares the memory with arr;

console.log(buf);

// Prints: <Buffer 88 13 a0 0f>

// changing the TypedArray changes the Buffer also

arr[1] = 6000;

console.log(buf);

// Prints: <Buffer 88 13 70 17>

The optional byteOffset and length arguments specify a memory range within the arrayBuffer that will be shared by the Buffer.

const ab = new ArrayBuffer(10);

const buf = Buffer.from(ab, 0, 2);

console.log(buf.length);

// Prints: 2

A TypeError will be thrown if arrayBuffer is not an ArrayBuffer.

### Class Method: Buffer.from(buffer)

* buffer {Buffer}

Copies the passed buffer data onto a new Buffer instance.

const buf1 = Buffer.from('buffer');

const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());

// 'auffer'

console.log(buf2.toString());

// 'buffer' (copy is not changed)

A TypeError will be thrown if buffer is not a Buffer.

### Class Method: Buffer.from(str[, encoding])

* str {String} String to encode.
* encoding {String} Encoding to use, Default: 'utf8'

Creates a new Buffer containing the given JavaScript string str. If provided, the encoding parameter identifies the character encoding. If not provided, encoding defaults to 'utf8'.

const buf1 = Buffer.from('this is a tést');

console.log(buf1.toString());

// prints: this is a tést

console.log(buf1.toString('ascii'));

// prints: this is a tC)st

const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');

console.log(buf2.toString());

// prints: this is a tést

A TypeError will be thrown if str is not a string.

### Class Method: Buffer.alloc(size[, fill[, encoding]])

* size {Number}
* fill {Value} Default: undefined
* encoding {String} Default: utf8

Allocates a new Buffer of size bytes. If fill is undefined, the Buffer will be *zero-filled*.

const buf = Buffer.alloc(5);

console.log(buf);

// <Buffer 00 00 00 00 00>

The size must be less than or equal to the value of require('buffer').kMaxLength (on 64-bit architectures, kMaxLength is (2^31)-1). Otherwise, a [RangeError][] is thrown. A zero-length Buffer will be created if a size less than or equal to 0 is specified.

If fill is specified, the allocated Buffer will be initialized by calling buf.fill(fill). See [buf.fill()][] for more information.

const buf = Buffer.alloc(5, 'a');

console.log(buf);

// <Buffer 61 61 61 61 61>

If both fill and encoding are specified, the allocated Buffer will be initialized by calling buf.fill(fill, encoding). For example:

const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');

console.log(buf);

// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>

Calling Buffer.alloc(size) can be significantly slower than the alternative Buffer.allocUnsafe(size) but ensures that the newly created Buffer instance contents will *never contain sensitive data*.

A TypeError will be thrown if size is not a number.

### Class Method: Buffer.allocUnsafe(size)

* size {Number}

Allocates a new *non-zero-filled* Buffer of size bytes. The size must be less than or equal to the value of require('buffer').kMaxLength (on 64-bit architectures, kMaxLength is (2^31)-1). Otherwise, a [RangeError][] is thrown. A zero-length Buffer will be created if a size less than or equal to 0 is specified.

The underlying memory for Buffer instances created in this way is *not initialized*. The contents of the newly created Buffer are unknown and *may contain sensitive data*. Use [buf.fill(0)][] to initialize such Buffer instances to zeroes.

const buf = Buffer.allocUnsafe(5);

console.log(buf);

// <Buffer 78 e0 82 02 01>

// (octets will be different, every time)

buf.fill(0);

console.log(buf);

// <Buffer 00 00 00 00 00>

A TypeError will be thrown if size is not a number.

Note that the Buffer module pre-allocates an internal Buffer instance of size Buffer.poolSize that is used as a pool for the fast allocation of new Buffer instances created using Buffer.allocUnsafe(size) (and the deprecated new Buffer(size) constructor) only when size is less than or equal to Buffer.poolSize >> 1 (floor of Buffer.poolSize divided by two). The default value of Buffer.poolSize is 8192 but can be modified.

Use of this pre-allocated internal memory pool is a key difference between calling Buffer.alloc(size, fill) vs. Buffer.allocUnsafe(size).fill(fill). Specifically, Buffer.alloc(size, fill) will *never* use the internal Buffer pool, while Buffer.allocUnsafe(size).fill(fill) *will* use the internal Buffer pool if size is less than or equal to half Buffer.poolSize. The difference is subtle but can be important when an application requires the additional performance that Buffer.allocUnsafe(size) provides.

### Class Method: Buffer.allocUnsafeSlow(size)

* size {Number}

Allocates a new *non-zero-filled* and non-pooled Buffer of size bytes. The size must be less than or equal to the value of require('buffer').kMaxLength (on 64-bit architectures, kMaxLength is (2^31)-1). Otherwise, a [RangeError][] is thrown. A zero-length Buffer will be created if a size less than or equal to 0 is specified.

The underlying memory for Buffer instances created in this way is *not initialized*. The contents of the newly created Buffer are unknown and *may contain sensitive data*. Use [buf.fill(0)][] to initialize such Buffer instances to zeroes.

When using Buffer.allocUnsafe() to allocate new Buffer instances, allocations under 4KB are, by default, sliced from a single pre-allocated Buffer. This allows applications to avoid the garbage collection overhead of creating many individually allocated Buffers. This approach improves both performance and memory usage by eliminating the need to track and cleanup as many Persistent objects.

However, in the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may be appropriate to create an un-pooled Buffer instance using Buffer.allocUnsafeSlow() then copy out the relevant bits.

// need to keep around a few small chunks of memory

const store = [];

socket.on('readable', () => {

const data = socket.read();

// allocate for retained data

const sb = Buffer.allocUnsafeSlow(10);

// copy the data into the new allocation

data.copy(sb, 0, 0, 10);

store.push(sb);

});

Use of Buffer.allocUnsafeSlow() should be used only as a last resort *after* a developer has observed undue memory retention in their applications.

A TypeError will be thrown if size is not a number.

### All the Rest

The rest of the Buffer API is exactly the same as in node.js. [See the docs](https://nodejs.org/api/buffer.html).

## Related links

* [Node.js issue: Buffer(number) is unsafe](https://github.com/nodejs/node/issues/4660)
* [Node.js Enhancement Proposal: Buffer.from/Buffer.alloc/Buffer.zalloc/Buffer() soft-deprecate](https://github.com/nodejs/node-eps/pull/4)

## Why is Buffer unsafe?

Today, the node.js Buffer constructor is overloaded to handle many different argument types like String, Array, Object, TypedArrayView (Uint8Array, etc.), ArrayBuffer, and also Number.

The API is optimized for convenience: you can throw any type at it, and it will try to do what you want.

Because the Buffer constructor is so powerful, you often see code like this:

// Convert UTF-8 strings to hex

function toHex (str) {

return new Buffer(str).toString('hex')

}

***But what happens if toHex is called with a Number argument?***

### Remote Memory Disclosure

If an attacker can make your program call the Buffer constructor with a Number argument, then they can make it allocate uninitialized memory from the node.js process. This could potentially disclose TLS private keys, user data, or database passwords.

When the Buffer constructor is passed a Number argument, it returns an **UNINITIALIZED** block of memory of the specified size. When you create a Buffer like this, you **MUST** overwrite the contents before returning it to the user.

From the [node.js docs](https://nodejs.org/api/buffer.html#buffer_new_buffer_size):

new Buffer(size)

* size Number

The underlying memory for Buffer instances created in this way is not initialized. **The contents of a newly created Buffer are unknown and could contain sensitive data.** Use buf.fill(0) to initialize a Buffer to zeroes.

(Emphasis our own.)

Whenever the programmer intended to create an uninitialized Buffer you often see code like this:

var buf = new Buffer(16)

// Immediately overwrite the uninitialized buffer with data from another buffer

for (var i = 0; i < buf.length; i++) {

buf[i] = otherBuf[i]

}

### Would this ever be a problem in real code?

Yes. It's surprisingly common to forget to check the type of your variables in a dynamically-typed language like JavaScript.

Usually the consequences of assuming the wrong type is that your program crashes with an uncaught exception. But the failure mode for forgetting to check the type of arguments to the Buffer constructor is more catastrophic.

Here's an example of a vulnerable service that takes a JSON payload and converts it to hex:

// Take a JSON payload {str: "some string"} and convert it to hex

var server = http.createServer(function (req, res) {

var data = ''

req.setEncoding('utf8')

req.on('data', function (chunk) {

data += chunk

})

req.on('end', function () {

var body = JSON.parse(data)

res.end(new Buffer(body.str).toString('hex'))

})

})

server.listen(8080)

In this example, an http client just has to send:

{

"str": 1000

}

and it will get back 1,000 bytes of uninitialized memory from the server.

This is a very serious bug. It's similar in severity to the [the Heartbleed bug](http://heartbleed.com/) that allowed disclosure of OpenSSL process memory by remote attackers.

### Which real-world packages were vulnerable?

#### [bittorrent-dht](https://www.npmjs.com/package/bittorrent-dht)

[Mathias Buus](https://github.com/mafintosh) and I ([Feross Aboukhadijeh](http://feross.org/)) found this issue in one of our own packages, [bittorrent-dht](https://www.npmjs.com/package/bittorrent-dht). The bug would allow anyone on the internet to send a series of messages to a user of bittorrent-dht and get them to reveal 20 bytes at a time of uninitialized memory from the node.js process.

Here's [the commit](https://github.com/feross/bittorrent-dht/commit/6c7da04025d5633699800a99ec3fbadf70ad35b8) that fixed it. We released a new fixed version, created a [Node Security Project disclosure](https://nodesecurity.io/advisories/68), and deprecated all vulnerable versions on npm so users will get a warning to upgrade to a newer version.

#### [ws](https://www.npmjs.com/package/ws)

That got us wondering if there were other vulnerable packages. Sure enough, within a short period of time, we found the same issue in [ws](https://www.npmjs.com/package/ws), the most popular WebSocket implementation in node.js.

If certain APIs were called with Number parameters instead of String or Buffer as expected, then uninitialized server memory would be disclosed to the remote peer.

These were the vulnerable methods:

socket.send(number)

socket.ping(number)

socket.pong(number)

Here's a vulnerable socket server with some echo functionality:

server.on('connection', function (socket) {

socket.on('message', function (message) {

message = JSON.parse(message)

if (message.type === 'echo') {

socket.send(message.data) // send back the user's message

}

})

})

socket.send(number) called on the server, will disclose server memory.

Here's [the release](https://github.com/websockets/ws/releases/tag/1.0.1) where the issue was fixed, with a more detailed explanation. Props to [Arnout Kazemier](https://github.com/3rd-Eden) for the quick fix. Here's the [Node Security Project disclosure](https://nodesecurity.io/advisories/67).

### What's the solution?

It's important that node.js offers a fast way to get memory otherwise performance-critical applications would needlessly get a lot slower.

But we need a better way to *signal our intent* as programmers. **When we want uninitialized memory, we should request it explicitly.**

Sensitive functionality should not be packed into a developer-friendly API that loosely accepts many different types. This type of API encourages the lazy practice of passing variables in without checking the type very carefully.

#### A new API: Buffer.allocUnsafe(number)

The functionality of creating buffers with uninitialized memory should be part of another API. We propose Buffer.allocUnsafe(number). This way, it's not part of an API that frequently gets user input of all sorts of different types passed into it.

var buf = Buffer.allocUnsafe(16) // careful, uninitialized memory!

// Immediately overwrite the uninitialized buffer with data from another buffer

for (var i = 0; i < buf.length; i++) {

buf[i] = otherBuf[i]

}

### How do we fix node.js core?

We sent [a PR to node.js core](https://github.com/nodejs/node/pull/4514) (merged as semver-major) which defends against one case:

var str = 16

new Buffer(str, 'utf8')

In this situation, it's implied that the programmer intended the first argument to be a string, since they passed an encoding as a second argument. Today, node.js will allocate uninitialized memory in the case of new Buffer(number, encoding), which is probably not what the programmer intended.

But this is only a partial solution, since if the programmer does new Buffer(variable) (without an encoding parameter) there's no way to know what they intended. If variable is sometimes a number, then uninitialized memory will sometimes be returned.

### What's the real long-term fix?

We could deprecate and remove new Buffer(number) and use Buffer.allocUnsafe(number) when we need uninitialized memory. But that would break 1000s of packages.

~~We believe the best solution is to:~~

~~1. Change new Buffer(number) to return safe, zeroed-out memory~~

~~2. Create a new API for creating uninitialized Buffers. We propose: Buffer.allocUnsafe(number)~~

#### Update

We now support adding three new APIs:

* Buffer.from(value) - convert from any type to a buffer
* Buffer.alloc(size) - create a zero-filled buffer
* Buffer.allocUnsafe(size) - create an uninitialized buffer with given size

This solves the core problem that affected ws and bittorrent-dht which is Buffer(variable) getting tricked into taking a number argument.

This way, existing code continues working and the impact on the npm ecosystem will be minimal. Over time, npm maintainers can migrate performance-critical code to use Buffer.allocUnsafe(number) instead of new Buffer(number).

### Conclusion

We think there's a serious design issue with the Buffer API as it exists today. It promotes insecure software by putting high-risk functionality into a convenient API with friendly "developer ergonomics".

This wasn't merely a theoretical exercise because we found the issue in some of the most popular npm packages.

Fortunately, there's an easy fix that can be applied today. Use safe-buffer in place of buffer.

var Buffer = require('safe-buffer').Buffer

Eventually, we hope that node.js core can switch to this new, safer behavior. We believe the impact on the ecosystem would be minimal since it's not a breaking change. Well-maintained, popular packages would be updated to use Buffer.alloc quickly, while older, insecure packages would magically become safe from this attack vector.

## links

* [Node.js PR: buffer: throw if both length and enc are passed](https://github.com/nodejs/node/pull/4514)
* [Node Security Project disclosure for ws](https://nodesecurity.io/advisories/67)
* [Node Security Project disclosure forbittorrent-dht](https://nodesecurity.io/advisories/68)

## credit

The original issues in bittorrent-dht ([disclosure](https://nodesecurity.io/advisories/68)) and ws ([disclosure](https://nodesecurity.io/advisories/67)) were discovered by [Mathias Buus](https://github.com/mafintosh) and [Feross Aboukhadijeh](http://feross.org/).

Thanks to [Adam Baldwin](https://github.com/evilpacket) for helping disclose these issues and for his work running the [Node Security Project](https://nodesecurity.io/).

Thanks to [John Hiesey](https://github.com/jhiesey) for proofreading this README and auditing the code.

## license

MIT. Copyright (C) [Feross Aboukhadijeh](http://feross.org)