View this article in Markdown | View this article on the web

# A Surprisingly Insidious Bug Reading Text in Chunks

*Using buffer.toString() on Node streams can silently corrupt UTF-8 when multibyte chars split across chunks... this is a rare bug, but can be hard to test, and even harder to reproduce.*

We need our system to read a file from the Internet. It's a *plain text* file (nothing could be simpler!), and then we need to parse, convert, summarize, normalize, reconcile, tabulate, save, transmit, spindle, mutilate, and/or otherwise process the file (which we think is going to be the *hard* part...)

We happen to be using Node, so I write a line or two of code in Node to fire off a network connection to go grab the file.

Now, we don't *always* or even *usually* want to read the whole file into memory all at once. The file might be large, and we could run out of memory loading the whole thing. The data might arrive slowly, and so we might want to process the data as it comes in instead of waiting until everything arrives to do anything.

Thus the usual Node approach to do is to get the data in chunks. The chunks are small enough to easily fit in memory, and we can process the chunks one at a time.

So, when I fire off my network connection, Node hands me chunks of the file as they arrive.

Since the file is supposed to be a *text* file, I could have set the option when opening the connection to have Node process the file as text for me, or have I could have set the encoding on the stream. That's the right way to do it. Node will do the conversion correctly if we ask it to. You can stop reading now!

Node doesn't do this by *default* because the file *could* be anything: it could be a PDF, a Word document, an executable program… Thus the default without setting the encoding is that the chunks arrive as what Node calls a "buffer", a purely binary chuck of bytes.

Maybe I didn't notice the option to set the encoding in the voluminous API documentation; maybe I've written code to open network connections thousands of times before and I don't stop to think that *this* time I could set the encoding; maybe I'm using a library that doesn't have an option to pass through the setting through to Node. Whatever the reason, now Node is handing me binary buffers.

My next step (I think) is to decode the buffer as plain text and get it into a string. I go reading the Node documentation for buffers, and there's a zillion methods, ah, but look, great, there's a "toString".

Or, these days, I might ask my favorite AI LLM model, "How do I convert a Node Buffer into a string?" Answer: "In Node.js, a Buffer is raw bytes. To turn it into a JavaScript string, use buffer.toString()". (This is the correct answer to the wrong question!)

```
stream.on('data', function (buffer) {
    process(buffer.toString());  // BUG!
}
```

Buffers contain bytes. A single byte isn't large enough to represent all the characters of all the languages of the world. (Not to mention emojis!) Many characters need more than one byte. So what happens if one of these multibyte characters juuuust happens to straddle two buffers? Now we've got just part of a character in the buffer. We can't successfully decode half a character.

Whoops.

For someone who already has in mind that multibyte characters can cross buffer boundaries, this code is *obviously*, *blatantly* wrong. For someone who doesn't have it in mind… it can be easy to overlook.

I've written code with this bug. I once asked an AI code agent to write a program to do some complex parsing and processing and it absolutely nailed it. Every complex conversion I asked it to do, it did correctly. But it started off by reading the text file... and oh look, it too has the multibyte character split across buffers bug.

This part isn't surprising. So far, this is just the ordinary kind of bug that appears in code all this time. I write code that has bugs, other people write code that has bugs, AI writes code with bugs.

For what makes this bug *insidious*, we need the rest of the story.

## Testing

After I write some code, I try it out. I point it to a text file on the Internet and see what I get. The same text I see when I look at the file myself. Good.

I could also write unit tests. As with all testing, there's a cost/benefit analysis involved. Writing unit tests cost engineering time. Not all unit tests are equal. It's unfortunately easy to write thousands of useless unit tests that all pass but don't test anything important and so don't guard against the kind of bugs that we actually encounter in production. For any particular set of unit tests we *could* write, we have to weigh the cost of writing and maintaining them compared the value of how much protection we expect them to give us against important bugs.

And honestly, this one line of code is so simple I probably wouldn't write tests for it. If I were *aware* of the multibyte character split across buffers bug I would *definitely* want to write a test for *that*. But if I were aware of the bug I wouldn't have written this code in the first place.

Let's say for the sake of argument I write tests. What sort of tests might I write? Well, what if we point the code at an *empty* file? A *small* file that entirely fits in one buffer? A *medium* sized file that fits in *two* buffers? A *large* file that needs *many* buffers?

Node will provide us a buffer with whatever data is *available*. Which might be a *small* buffer for a slow connection. Or a *large* buffer on a fast connection. So we

want to test not only small and large *files* but also small and large *buffers*.

The usual sizes for most buffers are powers of two: 1K, 2K, 4K, 16K, 32K, 64K… which are all an *even* number of bytes, but what if we got a buffer with an *odd* number of bytes? Would our code still work *then*?

And there are characters. There are simple characters, basic Unicode characters, complex Unicode characters, and super-duper really complex Unicode characters like the Sanskrit ka with virama forming kṣa, with udātta and candrabindu.

> A single character in Sanskrit:
>
> अ॒ग्निं

Given a particular set of unit tests running against a particular piece of code, how much additional confidence do those tests give us that the code is correct? For *some* kinds of code… concurrent code… indeterminate code… situations where the source code has been hacked to inject security vulnerabilities… the answer can be “little to none.”

However I’d expect unit tests to be useful for *this* code.

Of course, no amount of unit testing ever *proves* that code is correct. Yet we can still develop degrees of *confidence* in how *likely* we think that the code is correct. Low. Medium. High.

For this example with apparently comprehensive tests against some really simple code, I have as high a degree of confidence in the correctness of this code as I ever do for any typical commercially written code. Not to the standards required for medical devices or aviation code that might kill someone, but to what I usually see for code where if the site goes down it could cost the company millions of dollars.

What is different… unusual… strange… about *this* situation that my usual “best practices” fail so badly?

For that, we need a bit of history.

## History

In the earliest days of computers, characters were encoded with six bits. This was enough to represent the uppercase letters, the digits, and a few punctuation characters. To be clear, these were the uppercase *English* letters. This will become important shortly.

Technology marched on, and eventually we were able to use *seven* bits per character. This was large enough to include the *lowercase* English letters, and all the punctuation characters commonly found on typewriters! This encoding was called “ASCII”, and we still use it today.

Other countries with *non*-English languages had their own, *different* encodings.

Fast-forward a couple of decades, we came to the point where we could use multiple *bytes* for characters. This is enough to include *all* the world’s languages, and Unicode was born. And we got emojis.

By now all computers were using 8 bit bytes. If we’re using ASCII, how do we pack a 7 bit ASCII character into an 8 bit byte? Easy. Set the top bit to zero, and use the remaining 7 bits for the character.

The Unicode folks realized something clever. A modern file with 8-bit bytes containing only ASCII characters will have every top bit of every byte set to zero.

They came up with a Unicode encoding called “UTF-8” where if the top bit was *zero*, it encoded exactly the *same* character that ASCII did. And if it was *one*, this indicated a *multibyte* character for all of the *other* world languages.

This meant that if you were considering adapting Unicode and using UTF-8, all of your ASCII files still worked! Exactly as they were! No conversion! Every ASCII file in the world is also a UTF-8 file.

Now, my own keyboard is a *US* standard keyboard. If you look at the characters printed on the key caps themselves, the ones that I can type by pressing *one* key or “Shift” and one key, those are all *ASCII* characters. That’s not true for *other* countries. ASCII was a US standard for the English and punctuation characters used in the US.

I can of course type or generate other characters by pressing some more complex “Fn-Control-Option blah blah blah” keyboard chord or by copying and pasting. For what I write in *English* and what I can type *easily*, in UTF-8 these are all *single* byte characters because they came from ASCII.

A single byte character will never trigger the multibyte character split across buffers bug.

And, yes, I tested a bunch of different non-English characters in my unit tests. But I wasn’t thinking about “single byte characters” vs. “multibyte characters”. I was thinking about “English characters and non-English characters”. To catch the bug with a test, we need a non-English or other non-ASCII character **and** to have it positioned right at the chunk boundary to trigger the bug.

So now we have an *unpleasant*, *difficult*, and *painful* bug.

This however doesn’t yet quite rise to the level of *insidious*…

## The Outcome

My code goes into production, and at some point it loads a file with a multibyte character which straddles two chunks.

What happens next?

I can *imagine* a scenario that goes like this: the UTF-8 decoder hits the invalid character and throws an exception. The production error triggers an alarm so I go check the logs. I see that the original file contained correct UTF-8 and so we must have a decoding problem somewhere in our software. I fix the bug, restart processing, and the customer’s file goes through.

Well, no.

That’s not what happens.

You see, Unicode has a non-character called the “replacement character”.

If you happen to be viewing this article on the web and it isn’t being blocked by your browser etc., you might be able to see one right here at the end of this

sentence: 

If that didn't show up, here's an image:



If a UTF-8 decoder encounters a byte that's *invalid* for UTF-8, it's *replaced* by this "question mark in a black diamond" symbol.

There are *many* files out there which contain *mostly* ASCII, 99.9% ASCII, 99.99% ASCII, but have a *few* characters from one of the other hundreds of non-English encodings used before Unicode.

The UTF-8 decoder doesn't *throw away the entire file* just because it couldn't decode one or two characters.

If we open a file and we see gibberish with black diamonds and question marks all over the place, clearly something is wrong. But seeing the *occasional* replacement character? That's normal.

Unless we happen to *know* that the original file didn't contain any replacement characters, a file with a replacement character or two looks like an ordinary UTF-8 file. Some UTF-8 files contain a few replacement characters.

When might we see the bug? Depending on the Node version, how fast Node is loading the file, and so on, the chunk size might be end up being something like 64K. In the Markdown file containing the source for *this* article, we're only up to 11K right here. This file would need to be a *lot* longer for you to even have a *chance* of seeing corruption.

And, of course, the bug would only be triggered if there happened to be a multibyte character right at the chunk boundary.

Meanwhile chunks don't have a *fixed* size. Node provides what data is available. On a slower or faster connections, this could be a small chunk, a large chunk, or anything in between.

A customer complains that we corrupted their file. We try loading their file ourselves. We get a different chunk size, the chunk boundary happens not to

hit a multibyte character, and so no corruption. The customer tries again, different chunk size, no corruption. We close the bug as “not reproducible”.

To summarize, we have a bug which:

Is present in simple code that looks like it would be hard to get wrong… if we’re not already aware of the bug.

Can easily pass unit tests written to a high degree of comprehensiveness… again if if we’re not already aware of the bug.

Is triggered very rarely.

Doesn’t reproduce, unless we get particularly lucky.

Corrupts customer’s data.

And, doesn’t *look* like corrupted data if we’re not comparing it to the original.

Now *that* is an insidious bug.

## Code Review for a Rare Bug

This bug may be insidious, but it’s also rare.

The easiest, most straightforward way to read text data in Node is to set the encoding on the stream. Reading a UTF-8 text file? Set the encoding to “utf8”. Node takes care of the rest.

If for whatever reason we didn’t set the encoding and so Node is delivering binary buffers to us, we then need to *also* fail to realize that we can’t just convert the buffers to strings one by one.

In code review, the first step is to check whether the encoding is being set. If yes, we’re done, we don’t have this bug. (The code may of course have *other* bugs that prevent text from being read correctly, but not this *particular* bug.)

If the encoding isn’t being set… well, is there a reason why we *couldn’t* set the encoding? We can go down the path of custom buffer conversion code if we *have* to… but if we’re just reading text, that’s a high code review and testing

burden for something that could all be replaced with a call to Node's `setEncoding` method.

If, for some reason, we do need our own custom buffer conversion code, a unit test can plug in at the point where the code is being provided buffers by Node, and supply its own buffers. The unit test can then construct a couple buffers where a multibyte character straddles the two buffers, and check the text output.

---

Thank you for reading. Please email me at andrew@wilcoxsoftware.com if there's a technical topic you'd like to see written about, or if there's something I might be able to help you with on a contract basis.