19:06 Selectively Exceptional UTF8; or, Carefully tossing a spanner in the works.

by T. Goodspeed and R. Speers

In the good ol' days, software might be written once, in one programming language, with one parser for each file format. In the modern world, things can be considerably more complicated, with pieces of a complex distributed system using many programming language and databases, each with their own parsers. This is especially true in today's era of programming via deep stacks of libraries and frameworks, combined with proliferation of microservices, [39] it really matters how different languages treat what should be the exact same sequence of characters.

Sometimes it seems no one can agree on a character encoding scheme – the olde' ASCII ignores non-English languages, and since the internet realized the need for other language support, now developers consistently have to deal with frustrations like `str.encode('utf-16')` conversions between function calls. But, if everyone dropped their debates and adopted one standard – UTF-8,[40] UTF-16, or otherwise – we'd all finally be able to coexist – right?

Wrong. In this POC, we'll demonstrate how the differences between libraries and programming languages which parse the UTF-8 standard lead to inconsistent behaviors with parsing and recognition. We do *not* mean the numerous issues which have been previously discussed regarding making characters that look the same (homoglyphs),[41] file names which trick users to executing them,[42] or evading input filtering and validation.[43] Instead, we share parser differentials with how these libraries consume a sequence of bits, and interpret them as a set of UTF-8 commands.

A good starting point for these differentials would be to document differences in the *validity* of bytestrings as UTF-8, from the perspective of each language or library with which we might interact. Here we describe the validity of many such strings, grouping a number of UTF-8 implementations by their behavior when faced with tricky input.

In the context of this paper, a *string* means a string of bytes, rather than a decoded string of characters. A string is *tricky* if it is accepted by at least one interpreter and rejected by at least one other.

We present a number of bytestrings which are legal as UTF-8 in some but not all of eleven target implementations in programming languages and databases. Additionally, we present commentary and observations that might be useful in identifying other UTF-8 parser differentials and in exploiting those that are known.

A Quick Review of UTF-8

Out of many different standards for encoding text with characters unavailable in the ASCII standard, UTF-8 by Ken Thompson and Rob Pike became the dominant standard by 2009. Among other advantages, it is a superset of ASCII that can describe any codepoint available in the Unicode standard.

As of the Unicode Standard 6.0, UTF-8 consists of between one and four bytes that represent a codepoint between `U+0000` and `U+10FFFF`, with some regions such as `U+D800` to `U+DFFF` blacklisted. Bits are distributed as in Table 2, but further restrictions mean that only the sequences in Table 3 are considered to be well formed. We specify the version because these details have changed over time, with the standard being considerably more strict now than when it was first described.

---

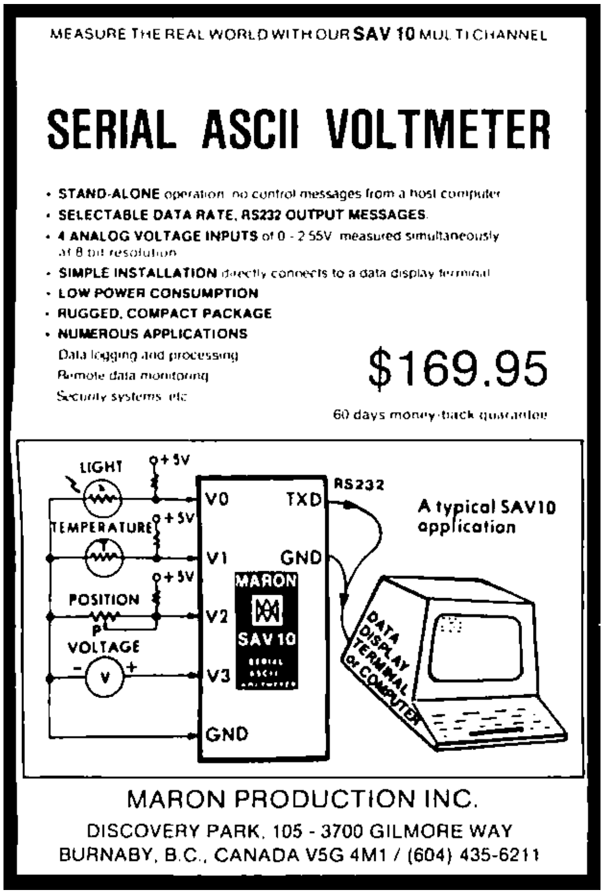
[39] A curated list of different micro-service frameworks across languages should convince the reader that this is not limited to a handful of languages.
`git clone https://github.com/mfornos/awesome-microservices`

[40] See RFC3629 - UTF-8, a transformation format of ISO 10646

[41] See references in Unicode Technical Report #36, or discussion of the internationalized domain name (IDN) homograph attack.

[42] This is a trick that malware authors have used to make the user see filenames like `happyexe.pdf`, but which is really `happyfdp.exe`.

[43] One example was MS09-20 (CVE-2009-1535) where "%c0%af" could be inserted into a protected path to bypass IIS's WebDAV path-based authentication system by making the path not match the authenticated rules list.



Plan9's early implementations of UTF-8 decoded to a 16-bit Rune, limiting UTF sequences to three bytes. There is no mention in Pike and Thompson's Usenix paper[44] of the forbidden surrogate pair range from `U+D800` to `U+DFFF`, and the three byte limit is understood to be a bit arbitrary.

For years, Windows has supported UTF-16 as wide characters (via the `wchar_t` type), but has used code page 1252 (similar to ANSI) for 8-bit characters. Internally there has been support for code page 65001 which is UTF-8, however it was not exposed until a build of Windows 10 as something that could be set as the locale code page.[45]

Similar Situations

As discussed in the introduction, we are not discussing the well-studied areas of homographs, other visual confusion, or filter evasion. Some prior work makes observations which have similarities, or hint at, the issues we discuss.

First, Unicode Technical Report #36 notes that in older Unicode standards, parsers were permitted to delete non-character code points, which led to issues when an earlier filter (e.g., a Web IDS) checked for some string like "exec(" that it didn't want to have present, but an attacker inserted an invalid code sequence in the string – so that it didn't match.[46] A different parser later in the stack may instead choose to delete this non-character code point, converting the string from "ex\uFEFFec(" to "exec(", thus possibly affecting the security of the application.

Similarly, the same document references issues that arise when systems compare text differently.[47] Similar situations are what we discuss here, however we focus on the string being judged as illegal, rather than compared differently, due to the parser differentials.

Blatantly Illegal Letters

Some sequences are blatantly illegal, and ought to be rejected by any decent interpreter. While we are most interested by the subtle differences between more modern interpreters, blatantly illegal characters are still useful in older languages, which might happily interpret them as bytestrings without attempting to parse them into runes.

As a general rule, older languages will only check the validity of a string if asked to. As a concrete example in Python 2, `"FB80808080".decode("hex")` will not trigger an exception, because the illegal string is only being interpreted as a string of bytes. `"FB80808080".decode("hex").decode("utf-8")` will trigger an exception, because the string is not legal in any reasonable UTF-8 dialect.

So when dealing with blatantly illegal strings, your difference of opinion might be found between a script that does check for validity and a second script *written in the same language* which does not.

---

[44] `unzip pocorgtof19.pdf utf.pdf`

[45] Insider build 17035 in November 2017.

[46] See clause "C7. When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-equivalent sequences *or the deletion of noncharacter code points*." (Emphasis added.)

[47] Unicode Technical Report #36 section 3.2

Ain't no law against bad handwriting.

Now that we've covered the theory, let's get down to some quirks of specific UTF-8 implementations. Follow along in Table 1 if you like.

Null Bytes

Null runes (`U+0000`) in UTF-8 are to be represented as a null byte (`00`), rather than encoded as a two-byte sequence (`C0 80`). Although Wikipedia mentions a "Modified UTF-8" that allows this sequence, in practice it has been rather hard for us to find one in surveying the major languages and libraries. All implementations that reject anything seem to reject the null pair.

What is worth noting, however, is that Postgres–perhaps only Postgres–will reject those strings which contain simple null bytes. You can express "`hello world\x00`" in nearly any other implementation, but perhaps for fear that naive C code might truncate it, Postgres will reject it.

```
psql (10.5 (Debian 10.5-1), server 9.6.7)
Type "help" for help.

user=> select E'hello\x00';
ERROR:  invalid byte sequence for encoding "UTF8": 0x00
user=>
```

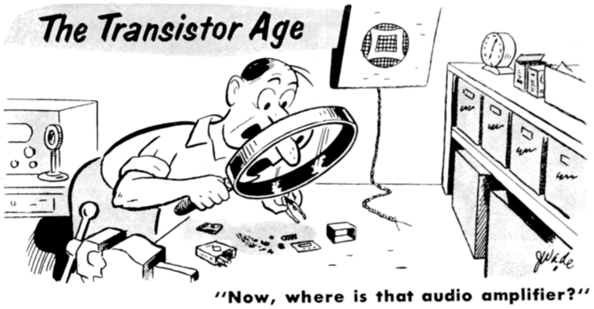
All other languages could care less.

```
Welcome to the MariaDB monitor.
Server version: 10.1.35-MariaDB-1 Debian unstable

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation
     Ab and others.

MariaDB [(none)]> select _utf8 X'3500';
+---------------+
| _utf8 X'3500' |
+---------------+
| 5             |
+---------------+
1 row in set (0.00 sec)

MariaDB [(none)]>
```



Surrogates

Some operating systems, such as Java and Windows, prefer to internally represent characters as 16-bit units. For this reason, UTF-16 uses pairs in the surrogate range from `D800` to `DFFF` to represent characters which use more than sixteen bits. This same range, `U+D800` to `U+DFFF`, is reserved in the Unicode standard so that no meaningful codepoints are excluded.

You can see in Table 1 that these surrogates are perfectly legal in Python 2 and MariaDB, but trigger exceptions in Python 3, Go, Rust, Perl 6, Java and .NET. Further experimentation with this would be handy, as surrogates can be either orphaned or in their proper, matching pairs.

Byte Counts

As we mentioned earlier, the pattern of UTF8 bit distribution shown in Figure 2 is very regular. An implementation could easily be restricted to three or four bytes by chance, and by continuing the pattern, one can easily imagine a fifth or sixth byte. In fact, implementations such as Perl 5 happily consume six byte UTF-8 runes, and a seven-byte implementation might be lurking in some interpreter, somewhere.

As a general rule, we see that ancient implementations support either three or six bytes, while the most modern languages seem to support four bytes. We've not yet found an implementation that supports only five bytes.

High Ranges

In addition to byte counts, implementations might disagree on the range within that number of bytes that they allow. Much like the surrogate range that we discussed earlier, the highest values of a range are sometimes restricted. These are the ranges that are missing from Table 3.

Where can we use this?

We argue that this isn't a theoretical issue. Indeed, it can arise in real-world software development projects.

One blog about micro-services hints at the issues someone will encounter during development with data representation, and the author does not discuss

---

[48] Blogger Richard Clayton wrote that "[w]e continuously encountered issues between the front and backend were serialization issues (UI using an Array, but Java expecting a String). While this isn't an issue specific to microservices, the problem is

security or character encoding differences.[48] The issues that such development teams feel is likely only the tip-of-the-iceberg if they were to start considering where differentials in the parsing of data representations could pose security or functionality issues.

Dodging the Logs

Companies routinely rely on logging and the indexing of these logs for use in debugging, optimization, security monitoring, and incident response. In the case of a web service, imagine one implemented in Python which presents a RESTful API that users interact with. To help determine when users act maliciously, all POST request activity is logged to a MariaDB database.

The `fourbyte` case presents a situation where the string `F0908D88h` is recognized and processed by the Python service, but if that same string is logged to a MariaDB or Postgres database, it will be treated as illegal and the insert would fail.

Disappearing Data

In another case, user input may be taken in, validated, and acted upon in one language, and then transferred to another system which rejects the string due to a parser differential. As we are not ones to advocate for keeping databases of everyone, especially not for minor misunderstandings of the speed limit, this could be handy in a hypothetical case where the drivers license database is maintained in one implementation, but where the speeding ticket database is implemented in a different language. Input to the speeding ticket database could come from the "trusted" license database, but fail to be processed and/or recorded in the ticketing system.

This may also be the case where a frontend written in one language has it's search index provided by another. One example may be Python frontend such as Reddit's legacy code[49] that uses Solr – a Java project – to provide search indexing. We haven't verified any such issues, and expanded cases would be needed to differentiate languages such as Python and Java.

Future steps for operations

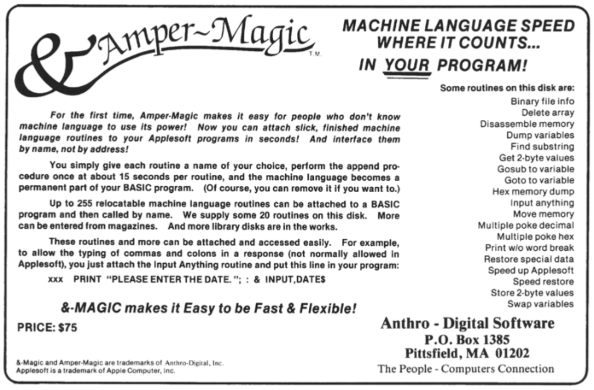
Someone looking to find vulnerable systems at scale will need to overcome a few challenges. First, the seemingly religious feud over mono-repos or multiple-repos means that modifying a project like `github-analysis`[50] to return statistics about *multiple* languages in a repository, as opposed to the primary one, is insufficient to identify many cases. If a repository, or set of them from one vendor, contains code in multiple languages, false positives (e.g., unit tests written in a different language, or dead code) need to be suppressed. Finally, dev-ops artifacts such as Dockerfiles, Cloud Formation scripts, and similar likely should be analyzed to identify third-party databases that are used. (Alternately code could be searched for database connection strings.)

We believe that future work to screen for projects where these bugs may exist will help bring this type of vulnerability to something which can be detected and mitigated.

Can everyone please agree already?

Of some hope for defenders is that Java, .NET, Python3, Go, Rust, and Perl 6 seem to all support very similar dialects, rejecting and accepting strings in step with one another.

We the authors therefore offer a bounty of a pint of good beer for each test case that newly differentiates these languages, by triggering an exception in one and not the others, up to a maximum of 64 beers.[51]



---

compounded when you increase the number of places these data representation issues can occur." `https://rclayton.silvrback.com/failing-at-microservices`

[49] `git clone https://github.com/reddit-archive/reddit`

[50] `git clone https://github.com/benfred/github-analysis`

[51] We the authors would also like to make clear that these will be excellent beers by our standards, but that Alexei Bulazel would consider them unworthy, as they are insufficiently valuable to be collateral in a mortgage, nor even for payment of a bridewealth or dowry.

		perl5	python2	python3 golang rust perl6 mono dotnet java	mariadb	postgres
surrogate	EDA081	1	1	0	1	0
nullsurrog	3000EDA081	1	1	0	1	0
threehigh	EDBFBF	1	1	0	1	0
fourbyte	F0908D88	1	1	1	0	0
fourbyte2	F0BFBFBF	1	1	1	0	0
fourhigh	F490BFBF	1	0	0	0	0
fivebyte	FB80808080	1	0	0	0	0
sixbyte	FD80808080	1	0	0	0	0
sixhigh	FDBFBFBFBF	1	0	0	0	0
nullbyte	3031320033	1	1	1	1	0

Table 1. Legality of Tricky UTF8 Strings in Five Dialects

`Scalar Unicode Value`	`First Byte`	`Second`	`Third`	`Fourth`
`00000000 00000000 0xxxxxxx`	`0xxxxxxx`			
`00000000 00000yyy yyxxxxxx`	`110yyyyy`	`10xxxxxx`		
`00000000 zzzzyyyy yyxxxxxx`	`1110zzzz`	`10yyyyyy`	`10xxxxxx`	
`000uuuuu zzzzyyyy yyxxxxxx`	`11110uuu`	`10uuzzzz`	`10yyyyyy`	`10xxxxxx`

Table 2. UTF-8 Bit Distribution, Unicode 6.0

`Scalar Unicode Value`	`First`	`Second`	`Third`	`Fourth`
`U+0000..U+007F`	`00..7F`			
`U+0080..U+07FF`	`C2..DF`	`80..BF`		
`U+0800..U+0FFF`	`E0`	`A0..BF`	`80..BF`	
`U+1000..U+CFFF`	`E1..EC`	`80..BF`	`80..BF`	
`U+D000..U+D7FF`	`ED`	`80..9F`	`80..BF`	
`U+E000..U+FFFF`	`EE..EF`	`80..BF`	`80..BF`	
`U+10000..U+3FFFF`	`F0`	`90..BF`	`80..BF`	
`U+40000..U+FFFFF`	`F1..F3`	`80..BF`	`80..BF`	`80..BF`
`U+100000..U+10FFFF`	`F4`	`80..8f`	`80..BF`	`80..BF`

Table 3. Well-Formed UTF-8 Byte Strings, Unicode 6.0