An Analysis of JSON Parsers in Rapid Web-Development Languages

Aarnav Mahavir Bos
aarnav.bos@code.berlin

CODE University of Applied Sciences

April. 2022

1 Abstract

This project conducts an evaluation of several JSON implementations in multiple langauges with a test-suite derived by the author. The JSON specification chosen for this project is the RFC 8259 by the IETF (Internet Engineering Task Force).

2 Introduction

JSON has quickly become the default standard for communication on the web due to it's lax, schema-less nature. Trends in web development show that often multiple applications are combined to compose one application or service. With multiple JSON specifications and extensions, utilized by a variety of implementations, it has become increasingly difficult to predict the validity of the data parsed in a multi-service architecture.

3 Evaluation Criteria

To narrow the scope of the project, four languages associated with rapid-development were chosen. They are Python, Node.js, Ruby and PHP. Rapid development often utilizes multiple third-party web services, and overlooks aspects such as data-integrity and security when combining services to iterate quickly. A total of fifteen JSON implementations were evaluated; chosen

in regards to their popularity. Streaming JSON parsers were not considered. Parserkiosk, an application written by the author that generates test cases for multiple languages from YAML(Yet Another Markup Language) was used in the evaluation.

The implementations are put through two hundred and eighty five test-cases, one hundred and ninety expect the parser to raise an error, and ninety-five are expected to be parsed successfully. The tests which are intended to pass also assert the language specific internal representation.

Language	Version
Python	3.10
Node.js	16.14.2
Ruby	ruby 2.7.4
PHP	7.4.28

Table 1: Chosen Languages

Implementation	Language	Is Standard Library
JSON	Node.js	Yes
json3	Node.js	No
json5	Node.js	No
json	Python	Yes
pysimdjson	C++ (Python bindings)	No
orjson	Python	No
rapidjson	C++ (Python bindings)	No
simplejson	Python	No
ujson	C (Python bindings)	No
JSON	Ruby	Yes
json_pure	Ruby	No
yajl	Ruby	No
oj	Ruby	No
json_decode	PHP	Yes
json5	PHP	No

Table 2: Chosen Implementations

4 Evaluation Results

Two hundred and twenty five tests failed across fifteen implementations. Thus denoting an average of fifteen tests failing per implementation.

Implementation	Language	Amount Failed
JSON	Node.js	6
json3	Node.js	6
json5	Node.js	6
json	Python	7
pysimdjson	C++ (Python bindings)	9
orjson	Python	6
rapidjson	C++ (Python bindings)	9
simplejson	Python	9
ujson	C (Python bindings)	26
JSON	Ruby	18
json_pure	Ruby	17
yajl	Ruby	48
oj	Ruby	30
json_decode	PHP	15
json5	PHP	15

Table 3: Results of test-suite per implementation

Language	Number of Implementations	Amount Failed
Python	6	64
Node.js	3	18
Ruby	4	113
PHP	2	30

Table 4: Results of test-suite per language

A matrix of the results (matrix.png) is included in the GitHub Repository. Additionally, a web-based matrix is also made available.

5 Margin of Error

With auto-generated test-cases testing multiple implementations over multiple languages, it is assumed that there will be false positives (tests that failed but shouldn't have).

6 Further Research

To further solidify the project, a special, small set of test-cases were written, explicitly looking for interoperability vulnerabilities.

6.1 Inconsistent Duplicate Key Precedence

When given duplicate JSON keys in an Object, the last member should be the accepted value. For example: `{"a": 1, "a": 2}` should parse to `{"a": 2}`

6.2 Permissive Parsing

Allowing trailing garbage in JSON.

```
{"a": 2}=
```

or allowing comments

```
{"a": 2, /*"a": 3*/}
```

or allowing strings without double quotes

```
{"a": string}
```

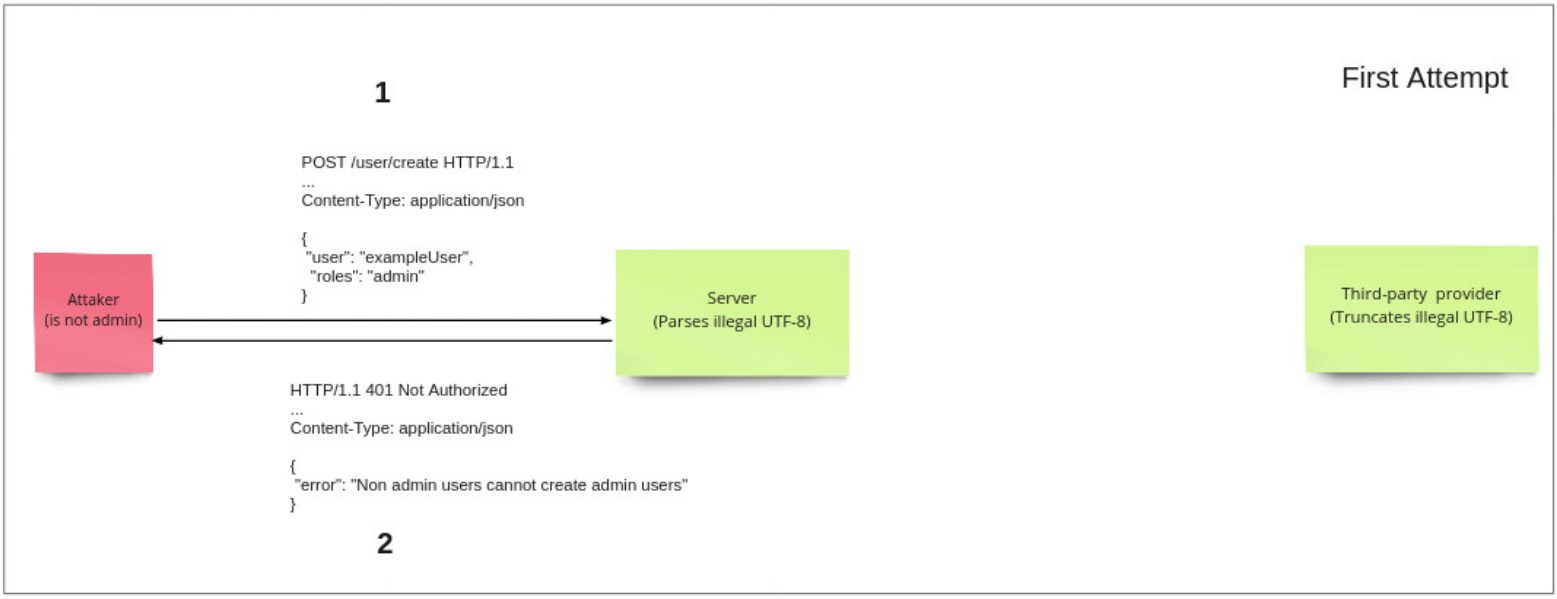
```
{"a": 'string'}
```

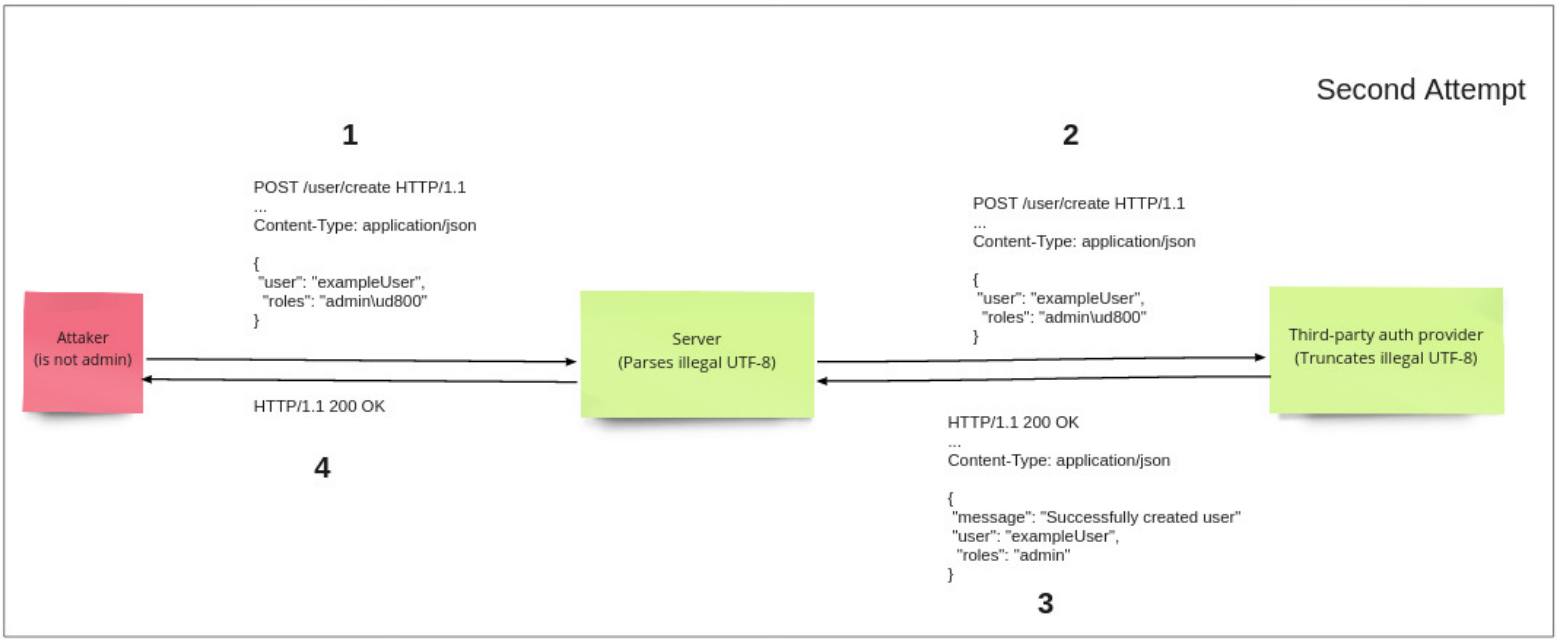
6.3 Illegal UTF-8 decoding/truncating

The possibility of decoding or truncating illegal UTF-8, instead of raising an error can lead to complex, persistent attacks.

For example:

This attack requires one service to decode illegal UTF-8 and another to truncate illegal UTF-8. Since `\ud800` is not valid UTF-8, it should raise an error, and not forward it.





The server runs the following check in `/user/create`

```
if current_user.role is not admin and user_to_create.role is admin:
    return 401
```

Since `"admin\ud800"` is not `"admin"` it will forward the request and successfully create an admin user.

7 Evaluation

The work in done the branch `further`
Nine tests were run looking for the vulnerabilies listed above. The only potential vulnerability was "Illegal UTF-8 decoding/truncating", with Ruby's yajl, Python's simplejson, Python's json and Python's ujson failing the test.

8 Mitigation Strategies

In regards to mitigation, if the application uses only internal services, a single parser implementation is encouraged. If that is not possible due to language constraints, an implementation like `rapidjson` is encouraged as it can be interfaced with many languages through FFI.

However, if an application uses external services, enforcing a strict implementation of the RFC 8259 is encouraged, thoroughly checking all JSON proxied through controlled services. Duplicate keys should raise an error as their unpredictability is very high. In general, string (UTF-8) parsing of the JSON parser should be thoroughly tested and fuzzed.

9 Conclusion

Parsing remains a volatile field, and has interoperability concerns that are evident. However, I wish to have had more time exploring the complexity of unicode, and write more test cases to further prove so.