

An Exploration of JSON Interoperability Vulnerabilities

Jake Miller

31-39 minutes

on Feb 25, 2021 5:00:00 AM

```
o = {  
  "qty": 1,  
  "qty": -1  
}  
o["qty"] // ???
```

TL;DR The same JSON document can be parsed with different values across microservices, leading to a variety of potential security risks. If you prefer a hands-on approach, [try the labs](#) and when they scare you, come back and read on.

INTRODUCTION: MORE PARSERS, MORE PROBLEMS

JSON is the backbone of web application communications. The simplicity of JSON is often taken for granted. We don't usually consider JSON parsing as part of our threat model. However, in our modern, multi-language, microservice architectures, our applications often rely on several separate JSON parsing implementations, each of which has its own quirks.

As we've seen through attacks like [HTTP request smuggling](#), discrepancies across parsers combined with multi-stage request processing can introduce serious vulnerabilities. In this research, I conducted a survey of 49 JSON parsers, cataloged their quirks, and present a variety of attack scenarios and [Docker Compose labs](#) to highlight their risks. Through our payment processing and user management examples, we will explore how JSON parsing inconsistencies can mask serious business logic vulnerabilities in otherwise benign code.

WHY ARE THERE PARSING INCONSISTENCIES?

Official and Alternative Specs

Even in the best-case implementation, there are inevitably minor, unintentional deviations from specifications. However, JSON parsers have a couple additional challenges. Even within the official JSON RFC, there is open-ended guidance on a few topics, such as how to handle duplicate keys and represent numbers. Although this guidance is followed by disclaimers about interoperability, most users of JSON parsers aren't aware of these caveats.

One contributing factor to inconsistencies among parsers is the differing specifications:

- [IETF JSON RFC \(8259 and prior\)](#): This is the official Internet

Engineering Task Force (IETF) specification.

- [ECMAScript Standard](#): Changes to JSON are released in lockstep with RFC releases, and the standard refers to the RFC for guidance on JSON. However, non-spec conveniences provided by the JavaScript interpreter, such as quoteless strings and comments, have inspired many parsers.
- [JSON5](#): This superset specification augments the official specification by explicitly adding convenience features (e.g., comments, alternative quotes, quoteless strings, trailing commas).
- [HJSON](#): HJSON is similar to JSON5 in spirit with different design choices.
- And more...

So, why would some parsers begin to selectively incorporate features that others ignore, or take contradicting approaches with parser behavior?

Open-ended Guidance

As discussed in the sections below, decisions on handling duplicate keys and representing numbers are often left open-ended. I suspect this may be due to the specification being published after implementations became popular. Perhaps the design committee decided not to break backwards compatibility with pre-specification JSON parsers, including the original JavaScript implementation.

However, these decisions continue to propagate through the ecosystem into super-set specs like JSON5 and HJSON and even into the binary variants like BSON, MessagePack, and CBOR, as we will discuss later on.

Further interoperability concerns come from delayed guidance on number and string encoding. String encoding was only explicitly required to be UTF-8 in the 2017 revision of the specification.

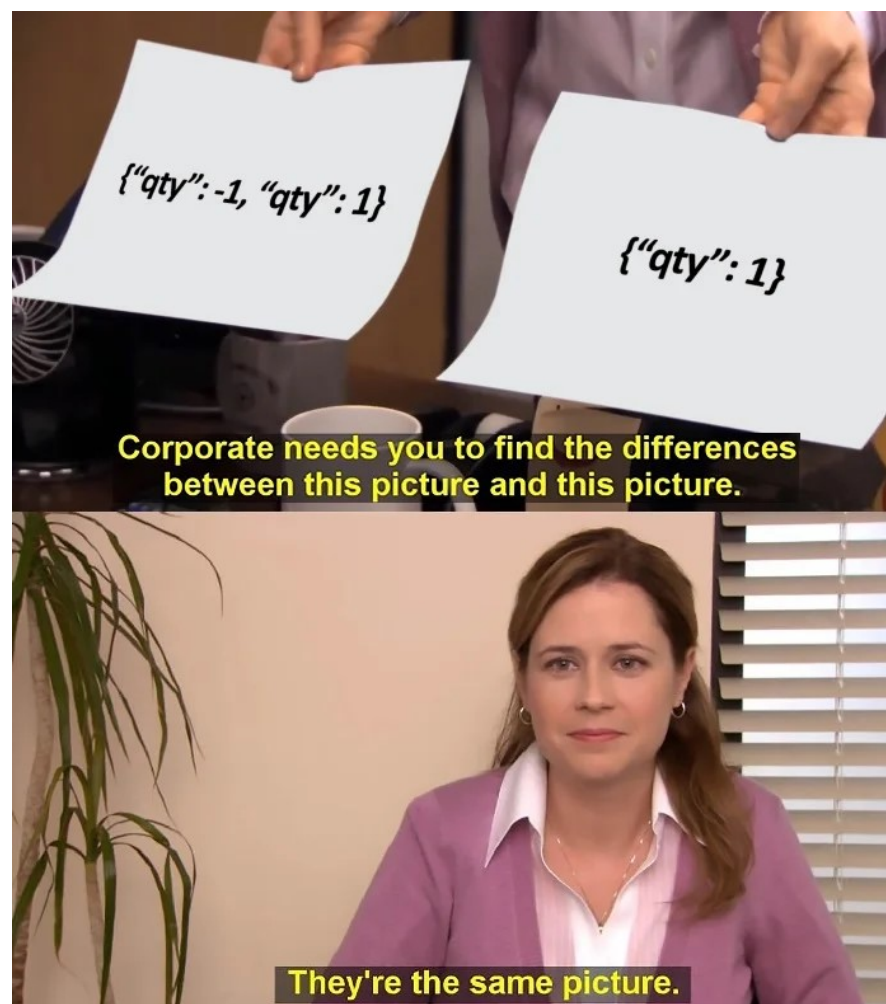
With that background context, let's explore what can go wrong.

JSON INTEROPERABILITY SECURITY RISKS

I categorized the identified interoperability security risks into five categories:

1. Inconsistent Duplicate Key Precedence
2. Key Collision: Character truncation and Comments
3. JSON Serialization Quirks
4. Float and Integer Representation
5. Permissive Parsing and Other Bugs

1. Inconsistent Duplicate Key Precedence



Creed said they were different. Creed and Pam were both right, but it depends on whom you ask.

Many of us have encountered this quirk of JSON through our development work: What happens if you have a duplicate key?

Consider the value of `obj["test"]` in the following document:

```
obj = { "test": 1, "test": 2 }
```

Is the value of `obj["test"]` going to be 1 or 2, or will it produce an error?

According to the official specification, any of those results are acceptable. Surprisingly, I've even encountered developers directly leveraging duplicate key precedence to create self-documenting JSON. Here's an example:

```
// For a parser where the last key takes  
precedence, the first "test" key will be ignored  
during parsing.  
obj = { "test": "this is a description of the test  
field", "test": "Actual Value" }
```

You may be surprised to find that the guidance from the spec is quite descriptive rather than prescriptive. The following is from the latest version of IETF JSON RFC (8259):

An object whose names are all unique is interoperable in the sense that all software implementations receiving that object will agree on the name-value mappings. When the names within an object are not unique, the behavior of software that receives such an object is unpredictable. Many implementations report the last name/value pair only. Other implementations report an error or fail to parse the object, and some implementations report all of the name/value pairs, including duplicates.

JSON parsing libraries have been observed to differ as to whether or not they make the ordering of object members visible to calling software. Implementations whose behavior does not depend on member ordering will be interoperable in the sense that they will not be affected by these differences.

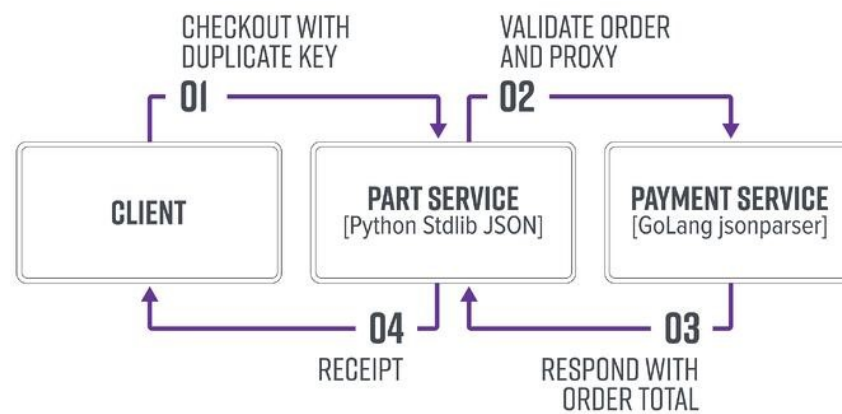
My suspicion is that the specification did not want to break backwards compatibility with pre-specification parsers. To be fair, the interoperability concerns are noted. But in a practical sense, as

stated earlier, how many developers read the JSON RFC, or consider interoperability issues with such a simple format? Needless to say, the language in the specification above is quite different than the explicit and direct guidance that is commonly found in an RFC.

So, let's look at some examples of how duplicate key precedence could go wrong.

Example: Validate-Proxy Pattern

Let's consider an e-commerce application where we have a Cart service that enforces business logic, forwards the request to a Payment service for payment processing, and performs order fulfillment. Let's attempt to get something for free. This example will use the design described below:



Let's suppose the Cart service receives a request like this (note the duplicated `qty` key for the second item in the cart):

```
POST /cart/checkout HTTP/1.1
...
Content-Type: application/json

{
    "orderId": 10,
    "paymentInfo": {
        //...
    },
    "shippingInfo": {
        //...
    },
    "cart": [
        {
            "id": 0,
            "qty": 5
        },
        {
            "id": 1,
            "qty": -1,
            "qty": 1
        }
    ]
}
```

As shown below, the Cart service enforces business logic before sending the order to the payment service. The API is written with Python Flask and uses the Python standard library JSON parser, which uses last-key precedence for duplicated keys (meaning for `id: 1` the `qty = 1`):

```
@app.route('/cart/checkout', methods=["POST"])
def checkout():
    # 1a: Parse JSON body using Python stdlib
    parser.
    data = request.get_json(force=True)
```

```

    # 1b: Validate constraints using jsonschema:
    id: 0 <= x <= 10 and qty: >= 1
    # See the full source code for the schema
    jsonschema.validate(instance=data,
schema=schema)

    # 2: Process payments
    resp = requests.request(method="POST",

url="http://payments:8000/process",
                                data=request.get_data(),
                                )

    # 3: Print receipt as a response, or produce
generic error message
    if resp.status_code == 200:
        receipt = "Receipt:\n"
        for item in data["cart"]:
            receipt += "{}x {} @
${}/unit\n".format(
                                item["qty"],
                                productDB[item["id"]].get("name"),
                                productDB[item["id"]].get("price")
                                )
            receipt += "\nTotal Charged:
${}\n".format(resp.json()["total"])
        return receipt
    return "Error during payment processing"

```

The JSON body will validate successfully because the duplicated key is ignored, and all the parsed values satisfy the constraints. Now that the JSON is deemed safe, the original JSON string (`request.get_data()`) is forwarded to the Payments service. From a developer's perspective, why would you waste computation by re-serializing a JSON object that you just parsed and validated when the string input is readily available? This assumption should be sound.

Next, the request is received at the Payments service. This Golang service uses a high-performance third-party JSON parser (`buger/jsonparser`). However, this JSON parser uses first-key precedence (meaning for `id: 1` the `qty = -1`). This service calculates the total, as shown below:

```

func processPayment(w http.ResponseWriter, r
*http.Request) {
    var total int64 = 0
    data, _ := ioutil.ReadAll(r.Body)
    jsonparser.ArrayEach(
        data,
        func(value []byte, dataType
jsonparser.ValueType, offset int, err error) {
            // Retrieves first instance of a
duplicated key. Including qty = -1
            id, _ := jsonparser.GetInt(value,
"id")
            qty, _ := jsonparser.GetInt(value,
"qty")
            total = total + productDB[id]
["price"].(int64) * qty;
        },
        "cart")

```

```
//... Process payment of value 'total'

// Return value of 'total' to Cart service for
receipt generation.
io.WriteString(w, fmt.Sprintf("{\"total\":
%d}", total))
}
```

The Cart service receives the total that was charged from the payment service and generates a receipt for the response. We view the receipt from Cart service, but we can see an error. We will be shipped six products of a value of \$700, but we were charged only \$300:

```
HTTP/1.1 200 OK
...
Content-Type: text/plain

Receipt:
5x Product A @ $100/unit
1x Product B @ $200/unit

Total Charged: $300
```

However, in this example, the validated JSON document was not re-stringified after being parsed; instead, it uses the JSON string from the original request. In [section 3, JSON Serialization Quirks](#), we will explore examples of re-stringified objects that still propagate risky quirks.

[Try out this attack in Lab 1.](#)

2. Key Collision: Character Truncation and Comments



It is also possible to induce key collisions through character truncation and comments, increasing the number of parsers affected by the duplicate key precedence.

Using Character Truncation

Some parsers truncate particular characters when they appear in a string, while others don't. This can cause different keys to be interpreted as duplicates in a subset of parsers. For example, the following documents will appear to have duplicate keys in some last-key precedence parsers, but not others:

```
{"test": 1, "test\[raw \x0d byte]": 2}
{"test": 1, "test\ud800": 2}
{"test": 1, "test'": 2}
{"test": 1, "te\st": 2}
```


These string representations are *usually* not stable for multiple rounds of deserialization and reserialization. For example, the Unicode codepoints U+D800 through U+DFFF are unpaired UTF-16 surrogates, and while unpaired surrogates can be encoded into a UTF-8 byte string, it's considered illegal Unicode.

All of these examples can be used in a similar manner to the previous example and Lab 1. However, environments that allow encoding and decoding of illegal Unicode (e.g., Python 2.x) may be susceptible to complex attacks that require storage (serialization) and retrieval (deserialization) of these values.

Let's start by observing the Unicode encoding and decoding behavior in Python 2.x:

```
$ python2
>>> import json
>>> import ujson

# Serialization into illegal unicode.
>>> u"asdf\ud800".encode("utf-8")
'asdf\xed\x80'

# Reserializing illegal unicode
>>> json.dumps({"test": "asdf\xed\x80"})
'{"test": "asdf\\ud800"}'

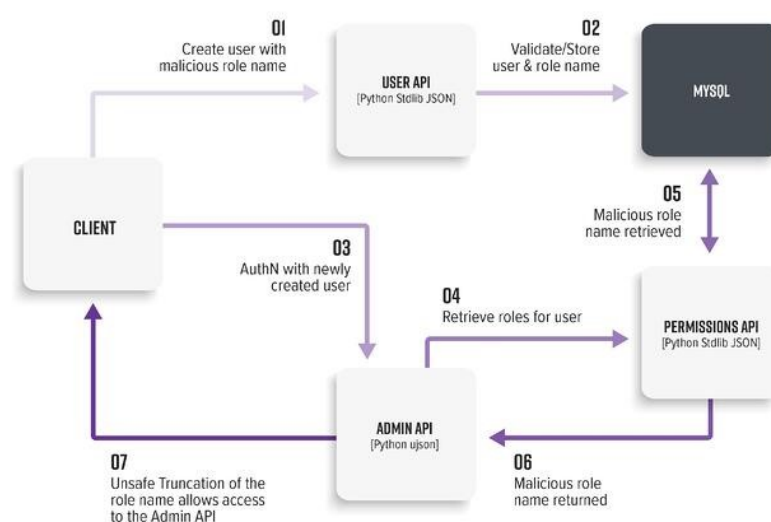
# Let's observe the third party parser ujson's
truncation behavior and how it creates a duplicate
key.
>>> ujson.loads('{"test": 1, "test\\ud800": 2}')
```

As we will see in our next example, an attacker can use this functionality to bypass sanitization checks, e.g., to create and store a role called `superadmin\ud888` that may be retrieved and parsed as `superadmin`. However, this technique requires support for encoding and decoding illegal Unicode codepoints (not so hard), as well as a database with a type system that won't throw exceptions (harder).

For the following lab, we will use Python 2.7 and MySQL in binary mode to allow us to focus on the risks of storing illegal unicode and its effect on inconsistent JSON parsing.

Example: Validate-Store Pattern

Let's consider a multi-tenant application where an organization admin is able to create custom user roles. Additionally, we know that users who have cross-organizational access are assigned the internal role `superadmin`. Let's attempt to escalate privileges. This example will use the design shown below:



First, let's try to force creating a user with `superadmin` privileges:

```
POST /user/create HTTP/1.1
...
```

```
Content-Type: application/json
```

```
{
  "user": "exampleUser",
  "roles": [
    "superadmin"
  ]
}
```

```
HTTP/1.1 401 Not Authorized
```

```
...
```

```
Content-Type: application/json
```

```
{"Error": "Assignment of internal role
'superadmin' is forbidden"}
```

As shown above, the User API has a server-side security control to block users from creating new users with the `superadmin` role.

This control is shared by the Roles API to avoid overwriting existing user-defined and system roles. Here, we'll assume our `/user/` and `/role/` endpoints on the User API use well-behaved, compliant parsers.

Instead, with the aim of affecting a downstream parser, we'll create a role with a name that is unstable across parsers,

```
superadmin\ud888:
```

```
POST /role/create HTTP/1.1
```

```
...
```

```
Content-Type: application/json
```

```
{
  "name": "superadmin\ud888"
}
```

```
HTTP/1.1 200 OK
```

```
...
```

```
Content-type: application/json
```

```
{"result": "OK: Created role 'superadmin\ud888'"}
```

Next, we create a user with the new user-defined role:

```
POST /user/create HTTP/1.1
```

```
...
```

```
Content-Type: application/json
```

```
{
  "user": "exampleUser",
  "roles": [
    "superadmin\ud888"
  ]
}
```

```
HTTP/1.1 200 OK
```

```
...
```

```
Content-Type: application/json
```

```
{"result": "OK: Created user 'exampleUser'"}
```

The User API stores the user into the database. Up until this point, all of the parsers see the user-defined role (`superadmin\ud888`) as a distinct name from the internal role `superadmin`.

However, when later accessing the cross-organizational `/admin` endpoint, the server requests the user's permissions from the Permissions API. The Permissions API faithfully encodes the

role, as shown below:

```
GET /permissions/exampleUser HTTP/1.1
...
```

```
HTTP/1.1 200 OK
...
Content-type: application/json
```

```
{
  "roles": [
    "superadmin\ud888"
  ]
}
```

But here is where things go wrong: The admin API uses the third-party `ujson` parser. As we saw earlier, this parser truncates any bytes that contain illegal codepoints:

```
@app.route('/admin')
def admin():
    username = request.cookies.get("username")
    if not username:
        return {"Error": "Specify username in Cookie"}

    username =
urllib.quote(os.path.basename(username))

    url = "http://permissions:5000/permissions
/{}".format(username)
    resp = requests.request(method="GET", url=url)

    # "superadmin\ud888" will be simplified to
    "superadmin"
    ret = ujson.loads(resp.text)

    if resp.status_code == 200:
        if "superadmin" in ret["roles"]:
            return {"OK": "Superadmin Access
granted"}
        else:
            e = u"Access denied. User has following
roles: {}".format(ret["roles"])
            return {"Error": e}, 401
    else:
        return {"Error": ret["Error"]}, 500
```

As shown above, our user-defined role will be truncated to `superadmin`, granting access to the privileged API.

[Try out this example in Lab 2.](#)

Many JSON libraries support quoteless strings and comment syntax from JavaScript interpreter environments (e.g., `/*, */`).

However, neither of these features is part of the official specification. These features allow parsers to process documents like the following:

```
obj = {"test": valWithoutQuotes, keyWithoutQuotes:
"test" /* Comment support */}
```

Given two parsers that support quoteless strings, but only one that recognizes comments, we can smuggle duplicate keys. Consider the example below:

```
obj = {"description": "Duplicate with comments",
"test": 2, "extra": /*, "test": 1, "extra2": */}
```

Here we will use the serializer from each parser to view its respective output.

Serializer 1 (e.g., GoLang's GoJay library) will produce:

- `description = "Duplicate with comments"`
- `test = 2`
- `extra = ""`

Serializer 2 (e.g., Java's JSON-iterator library) will produce:

- `description = "Duplicate with comments"`
- `extra = "/*"`
- `extra2 = "*/"`
- `test = 1`

Alternatively, straightforward use of comments can also be effective:

```
obj = {"description": "Comment support", "test": 1, "extra": "a"/*, "test": 2, "extra2": "b"*/}
```

Here is the decoding performed by Java's GSON library:

```
{"description":"Comment support", "test":1, "extra":"a"}
```

And here is the decoding performed by Ruby's simjson library:

```
{"description":"Comment support", "test":2, "extra":"a", "extra2":"b"}
```

This demonstrates how inconsistencies in additional parsing features can enable unique key collision attacks.

3. JSON Serialization Quirks



So far, we've only focused on JSON decoding, but nearly all implementations also offer JSON encoding (aka serialization). Let's look at a few examples.

Inconsistent Precedence: Deserialization vs. Serialization

Conventional wisdom is to avoid duplicate keys, which is easy to do with internal services, but not possible to guarantee with external user input. So, not all parsers explore the behavior with duplicate keys. In one instance, a parser (Java's JSON-iterator) produced the following input and outputs:

Input:

```
obj = {"test": 1, "test": 2}
```

Output:

```
obj["test"] // 1
obj.toString() // {"test": 2}
```

As shown above, the values of key retrieval and serialization differ. The underlying data structure seems to have preserved the value of the duplicate key; however, the precedence between the serializer and deserializer was inconsistent.

Generating Documents with Duplicate Keys

Per the specification, it's acceptable to serialize duplicate keys, and some parsers (e.g., C++'s rapidjson) do just that:

Input:

```
obj = {"test": 1, "test": 2}
```

Output:

```
obj["test"] // 2
obj.toString() // {"test": 1, "test": 2}
```

In these cases, reserializing parsed JSON objects does not provide protection. These serialization behaviors allow attackers to smuggle values across sanitization layers. As we saw previously, this can lead to business logic flaws, injection vulnerabilities, or other security impacts.

4. Float and Integer Representation



Now that we've observed the many risks of duplicate keys, we'll examine number representation. First, let's look at an excerpt of the RFC discussing number interoperability:

Since software that implements IEEE 754 binary64 (double precision) numbers [IEEE754] is generally available and widely used, good interoperability can be achieved by implementations that expect no more precision or range than these provide, in the sense that implementations will approximate JSON numbers within the expected precision. A JSON number such as 1E400 or 3.141592653589793238462643383279 may indicate potential interoperability problems, since it suggests that the software that created it expects receiving software to have greater capabilities for numeric magnitude and precision than is widely available.

Note that when such software is used, numbers that are integers and are in the range $[-(2^{53})+1, (2^{53})-1]$ are interoperable in the sense that implementations will agree exactly on their numeric values.

Let's start by looking at large numbers.

Inconsistent Large Number Decoding

When not decoded accurately, large numbers may be decoded as `MAX_INT` or 0 (or `MIN_INT` as we approach negative infinity).

Across multiple parsers, we will find that a large number, such as:

can be decoded to multiple representations, including:

We'll try out an example.

Let's revisit Lab 1. We know that the third-party Golang jsonparser library used in the Payments API will decode large numbers to 0, while the Cart API will decode the number faithfully. We can exploit this inconsistency to get free items. Let's purchase a large quantity of e-gift cards (`id: 8`):

```
POST /cart/checkout HTTP/1.1
...
Content-Type: application/json
```

Response:

[illegible]

Total Charged: \$0

The business logic layer faithfully decodes the integer, while the payment processing layer defaults to the value 0 for large numbers.

[Try this attack out in Lab 1: part 2](#) with the request in `lab1_alt_req.json`. As described in the lab, the `jsonparser` library can detect this overflow with proper error checking.

Positive and negative infinity along with NaN (not a number) are not supported by the official RFC. But many parsers have chosen workarounds. Deserializing and/or reserializing values can lead to a

variety of outcomes such as:

Input:

```
{"description": "Big float", "test": 1.0e4096}
```

Output:

```
{"description": "Big float", "test": 1.0e4096}
{"description": "Big float", "test": Infinity}
{"description": "Big float", "test": "+Infinity"}
{"description": "Big float", "test": null}
{"description": "Big float", "test": Inf}
{"description": "Big
float", "test": 3.0e14159265358979323846}
{"description": "Big
float", "test": 9.218868437227405E+18}
```


Note the type conversions from a JSON number to a string. Type conversions in strict comparison can be benign but in loose comparisons can lead to type juggling vulnerabilities. Consider the following code (note: strings are interpreted as 0):

```
<?php
echo 0 == 1.0e4096 ? "True": "False" . "\n"; #
False
echo 0 == "Infinity" ? "True": "False" . "\n"; #
True
?>
```

As in previous examples, business logic layers may falsely validate values that are decoded inconsistently. Prefer strict comparison or perform type validation prior to usage.

5. Permissive Parsing and One-off Bugs

```
010001100110010101100001011101000110101011100
100100101011100110010000001110011011101010110
0011011010000010001011100110010000001
100001011011000111101011100100110111
001100001011101000101010110001101101
0000010000001000010000001100001011
01100011101000110011100110111001100001
011100110010000001110011011101010110001101101
000001000000100001011100110010000001100001011
```



Some parsers were permissive of stray characters, alternative quote characters, and syntax errors in documents, while others had strict enforcement of the RFC-defined grammar. Let's look at instances of permissive parsing that do not relate to duplicate keys.

Trailing Garbage

Allowing trailing garbage is a well-known issue with many JSON parsers that has been misused for many years to conduct cross-site request forgery (CSRF) attacks. To bypass same-origin policy (SOP) "simple request" limitations, a JSON document can be posted with a trailing equals sign to suggest a `x-form-urlencoded` document, which is permitted in cross-origin requests. For example:

```
POST / HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded

{"test": 1}=
```

Services that disregard the `Content-Type` and process all requests as JSON will be exposed to these types of CSRF attacks.

Denial-of-Service: Segmentation Faults

Two parsing libraries crashed on malformed JSON. Both of these instances have been reported to the maintainers. The names of the affected parsers will be updated in the future following remediation.

That said, in the interest of performance, many parsers rely on low-level routines that may be susceptible to memory safety issues. Be sure to keep an eye on parsers that rely on native code. Joining programs like Google's OSS-Fuzz is a great way to get easy access to fuzzing.

WHAT ABOUT BINARY JSON PARSERS

I briefly tested BSON, MessagePack, UBJSON, and CBOR formats, and a sampling of their respective parsers. Many of the same issues persisted for these parsers as well. Although some serializers refused to create binary representations of multiple keys, we can manually create malicious documents by byte swapping, as shown below for MessagePack:

```
json_doc = {u'test': 1, u'four': 2} # use an ABI-compatible string to replace (e.g., 'four')
encoded = msgpack.packb(json_doc,
use_bin_type=True)
encoded = encoded.replace(b'four', b'test')
```

Next, we process this document with the Dot Net MessagePack deserializer and use the included JSON serializer:

```
Console.WriteLine(MessagePackSerializer.ConvertToJson(File.ReadAllBytes("msgpack.bi
```

This produced the following output:

```
{"test":1,"test":2}
```

If we look at the documentation, many of the specifications try to be backwards compatible with JSON. That said, specs like BSON do attempt to be more explicit about how to handle situations like duplicate keys.

There are certainly more opportunities to discover further interoperability issues in these binary formats.

RESULTS: PARSER BEHAVIOR

The survey consisted of 49 parsers across the following languages, representing both standard library parsers (when available) and third-party parsers:

- C/C++
- C#
- Elixir/Erlang
- Go
- Java
- JavaScript
- PHP
- Python
- Ruby
- Rust

(Please refer to Appendix A for complete list of parsers and version numbers.)

The results below define parsers with uncommon behavior. These represent deviations from the norm and/or the official specification. Some of this behavior is a deliberate design choice and defined by the respective super-specifications (e.g., JSON5).

Other than segmentation faults, these behaviors are harmless in the context of a single parser, which prevents them from being classified as vulnerabilities for a particular parser. However, as we've observed, the behavior can introduce security risks through interoperability. As such, the following behaviors may or may not change in the future:

First-key precedence for duplicate keys:

- Go[jsonparser]
- Go[gojay]
- C++[rapidjson]
- Java[json-iterator]
- Elixir[Jason]
- Elixir[Poisson]
- Erlang[jsone]

Character truncation:

- Collision through unpaired surrogate
- Python[ujson]
- PHP [json5]
- Collision through backslash followed by carriage return byte (0x0d)
- Rust[json5]
- PHP[json5]
- Collision through stray quotes
- Ruby[simdjson]
- Collision through stray backslash { "test": 2, "te\st": 1 }
- C#[Jayrock.json]
- Ruby[stdlib/ext]
- Ruby[stdlib/pure]
- JavaScript[json5]
- Rust[json5]

Denial-of-service (segmentation fault):

- (Two instances) To be announced following remediation

Comment truncation:

- Java[json-iterator]
- Ruby[simdjson]

Stringified duplicate keys:

- C++[rapidjson]
- C#[MessagePack] (v2.2.85)

Unexpected comment support (non-JSON5/HJSON parsers):

- Ruby[stdlib/ext]
- Ruby[stdlib/pure]
- Ruby[oj]
- Ruby[Yajl]
- Go[jsonparser]
- Go[gojay]

- Java[GSON]
- Java[Genson]
- Java[fastjson]
- C#[Newtonsoft.Json]
- C#[Utf8Json]
- C#[Jayrock.Json]
- C#[Manatee.Json]

Large numbers (converted to string or “Infinity”):

- Python[stdlib/json] – In: 1.0e4096, Out: “Infinity”
- Python[ujson] In: 1.0e4096, Out: “Inf”
- Java[Jackson] – In: 1.0e4096, Out: “Infinity”
- Java[Genson] – In: 1.0e4096, Out: “Infinity”
- Java[Jodd] – In: 1.0e4096, Out: “+Infinity”
- C#[Manatee] – In: 1.0e4096, Out: “Infinity”
- C#[Newtonsoft.Json] – In: [9 repeated 96 times] Out: “[9 repeated 96 times]”

Large numbers (significant rounding):

- Ruby[oj] – In: 1.0e4096, Out: 3.0e14159265358979323846
- C#[Utf8Json] – In: 1.0e4096, Out: 9.218868437227405E+18
- PHP[jsonlint] – In: [9 repeated 96 times]
Out:9223372036854775807

Large numbers (returns 0 without error checking):

- Go[jsonparser] – In: 1.0e4096, Out: 0,

Overall, I found that, of the 49 JSON parsers surveyed, each language had at least one parser that exhibited a form of potentially risky interoperability behavior. Parsers provided by standard libraries tended to be the most compliant, but they often lacked speed, which is of increasing importance in microservice architectures. This has prompted developers to choose more performant, third-party parsers.

Back when JSON was just a part of the ECMAScript standard, it would have been hard to know how widespread it would become. There are some great lessons to be learned from this RFC and the interoperability bugs discussed above. I have outlined remediation and testing guidance for different audiences below:

For JSON parser maintainers:

- Generate fatal parse errors on duplicate keys.
- Do not perform character truncation. Instead, replace invalid Unicode with placeholder characters (e.g., unpaired surrogates should be displayed as the Unicode replacement character U+FFFD). Truncating may break sanitization routines for multi-parser applications.
- Avoid deviating from the chosen specification, or provide a "strict" mode that adheres to the RFC 8259 (or relevant) definition.
- Produce errors when handling integers or floating-point numbers that cannot be represented faithfully.

For software engineers:

- Take inventory of your existing parsers across your architecture. Identify behavior gaps in the parsers using the provided test-cases, available documentation, or against the existing list of behaviors above.

For security folks:

- These are nuanced attacks and not easy to identify from the

outside. If you have access to source code, look for parsers with known quirks. Try duplicating keys and using the suggestions in the [Labs README](#) to try to induce collisions.

WHAT ABOUT: JSON SCHEMA VALIDATORS?

The JSON Schema specification can help simplify and enforce type-safety and constraints, but it can't help with duplicate keys. JSON Schema implementations do not perform JSON parsing themselves, but instead process only the parsed object. To demonstrate this point, I used JSON Schema in the vulnerable labs that accompany this blog post.

For example, Java's JSON Schema implementation requires `org.json.JSONObject` inputs and Python's `json-schema` library relies on Python Dictionary objects, as shown below:

```
import jsonschema

schema = {
    "type": "object",
    "properties": {
        "test": {
            "type": "integer",
            "minimum": 0,
            "maximum": 100,
        }
    }
}

jsonschema.validate(instance={"test": 1},
schema=schema)
```

JSON Schema may help mitigate some parsing risks, such as type checking and constraining the range of allowed integers. But inconsistent parsing will be a blind spot for JSON Schema.

WHAT ABOUT JSON LINTERS?

JSON linters consume string input and are often used for IDEs or a programming environment. Linters are just parsers that usually don't return a decoded object. Ideally, we want consistent parsing with our decoding parsers.

TAKEAWAYS

Parsing discrepancies will continue to be a theme in microservice security. Although the conditions for these attacks may be challenging to detect, the labs demonstrate that even seemingly benign standards like JSON can introduce unexpected interoperability quirks that lead to serious security issues in more complex systems.

Finally, when designing protocols or standards, restricting behavior to deterministic outcomes not only improves interoperability but also makes it easier to report bugs and improve our software. Breaking backwards compatibility in existing standards by defining previously undefined behavior may cause pushback. But in the modern context of microservice architectures, where interoperability becomes increasingly complex, it may be a worthwhile choice.

UPDATE 02/26/21: Thank you InfoSec Twitter for bringing another awesome resource on JSON parsing quirks to my attention: http://seriot.ch/parsing_json.php (2018) by Nicolas Seriot (@nst021). Be sure to check it out for a deep-dive on JSON parsing. It also includes parser quirks for languages not covered in this article (e.g., Perl, Lua, Swift, and more).

APPENDIX A: VERSION NUMBERS

| | | |
|---|--|---|
| <ul style="list-style-type: none"> • C/C++ • tencent/rapidjson 1.1.0 • nholmann/json 3.9.1 • C# • Text.Json (Runtime 5.0.1) • Json 12.0.3 • Utf8Json 1.3.7 • Json 0.9.16530.1 • Json 13.0.4 • Elixir/Erlang • jason 1.2 • poison 4.0 • jiffy 1.0 • json 1.4 • jsone 1.5 • jsx 3.0 • Go • encoding/json (go1.14.7) • buger/jsonparser (cb835d480ac58e1b4be76afeac49e89ed651c3b5 Jul 30 2020) • francoispqt/gojay (v1.2.13) | <ul style="list-style-type: none"> • Java • json 20201115 • Jackson-databind 2.12.0 • Genson 1.6 • Jodd 6.0.2 • fastjson 1.2.73 • json-iterator 0.9.23 • JavaScript • JSON (stdlib) • json5 2.1.3 • json-buffer 3.0.1 • buffer-json 2.0.0 • parse-json 1.3.1 • secure-json-parse 2.1.0 • PHP • json_decode (stdlib) • colinodell/json5 2.1.0 • seld/jsonlint 1.8.2 • halaxa/json-machine 0.3.3 • salsify/jsonstreamingparser 8.2.0 | <ul style="list-style-type: none"> • Python 3 • json (stdlib) • orjson 3.4.6 • pysimdjson 3.1.1 • rapidjson 1.0 • simplejson 3.17.2 • ujson 4.0.1 • Ruby • JSON ext (stdlib; Ruby 2.7.2p137) • JSON pure (stdlib; Ruby 2.7.2p137) • json5 0.0.1 • oj 3.10.18 • yajl-ruby 1.4.1 • simdjson 1.0.1 • Rust • serde_json 1.0.60 • json 0.12.4 • json5 0.3.0 • serde-hjson 0.9.1 |
|---|--|---|