Modeling a JSON data spill

Roger L. Costello
July 25, 2017

# Introduction

Here is a JSON object:

{
 "title": "Classified information",
 "title": "The Adventures of Huckleberry Finn"
}

That JSON object has two "title" keys. The JSON object has duplicate keys.

JSON parsers have been tested and found to have varying behavior with JSON objects containing duplicate keys. Some parsers ignore the first key and expose just the last key. One could imagine a bad guy exploiting such a parser’s behavior, concealing classified information in the first key.

The JSON specification (RFC 7159) does not specify how duplicate keys should be handled:

“When the names within an object are not unique, the behavior of software that receives such an object is unpredictable.”

Suppose an application receives the above JSON instance document, parses it using the JSON parser, evaluates the data exposed by the parser, and releases the JSON instance document since "The Adventures of Huckleberry Finn" is unclassified. That is a data spill (data disclosure).

# Desire a more formal description

The description in the previous section is an intuitive explanation of how a data spill can occur. We desire a more formal description. The notation used in this paper is the Alloy notation [[Alloy](http://alloy.mit.edu/alloy/)]. Using the Alloy notation, we will model a JSON parser and an application that uses the parser. We then use the Alloy Analyzer to analyze the model and show how the the application generates results containing classified data.

# This is the scenario we will model

A JSON instance document contains a JSON object. The object contains classified and unclassified data. The instance document is input to a JSON parser. The parser processes the JSON object, outputting (exposing) a set of keys. The application uses the output of the parser to generate a result. The application removes keys in the parser’s output that contain classified data.

The graphic below shows the components that make up the model – a JSON object is input to the parser; an application lies on top of the parser, taking its output to generate a result.



# Parser model

The parser takes JSON as input and generates output. This model focusses just on JSON object input (not JSON arrays, JSON strings, or other JSON values). The output of the parser is the set of keys (name/value pairs) in the input object.

**one sig** Parser {

 input: Object,

 output: **set** Key

} {

 // The input JSON object consists of all of the keys in the set Key.

 Key **in** input.keys

}

A JSON object consists of a set of keys.

**sig** Object {

 keys: **set** Key

}

Each key has a name and a value (data).

**sig** Key {

 name: Name,

 value: Data

}

Set of names.

**sig** Name {}

Data may be either classified or unclassified.

**abstract** **sig** Data {}

**sig** Unclassified **extends** Data {}

**sig** Classified **extends** Data {}

“parse” is the parsing operation. After initialization, each key in the input is sequentially processed. A key is added to the ouput only if it is the final key in the object with that name. In other words, when duplicate keys exist, only the last one is exposed (output).

**pred** parse {

 initParser

 **all** key: Parser.input.keys |

 **let** nextKeys = nexts [key] |

 **not** (key.name **in** nextKeys.name) =>

 key **in** Parser.output

 **else**

 **not** (key **in** Parser.output)

}

To make it interesting, the input object contains at least one key and not all keys have the same name.

**pred** initParser {

 **some** Parser.input.keys

 #Name > 1

}

# Application model

The application first invokes the parser. It processes the keys output by the parser, discarding those with classified values, retaining those with unclassified values. The result of the application is a new JSON object.

There are a set of Applications, each representing a state of the Application.

**sig** Application {
 result: Object
} {
 // The JSON object output by the application is not the same as that input to the parser.
 result != Parser.input
}

The application uses the following execution strategy: start with the original JSON object keys and iterate over the keys output by the parser; if a key has a classified value, remove it; otherwise, include it in the the next state of the Application.

**fact** trace {
 init [first]
 **all** app: Application - last | **let** app' = app.next {
 **all** key: Parser.output |
 key.value **in** Classified =>
 app'.result.keys = app.result.keys - key
 **else**
 app'.result.keys = app.result.keys
 }
}

Initialize the Application's result field to hold the original JSON object. Then call the parser.

**pred** init (app: Application) {
 app.result.keys = Parser.input.keys
 parse
}

Okay! That’s the model, for both the parser and the application. Now it’s time to analyze the model. Of particular interest is whether the model allows the application’s result to contain classified data. We assert that it doesn’t and invite the Alloy Analyzer to find counterexamples.

Assert: The application’s result has no keys with classified data.

**assert** No\_Classified\_Data\_In\_App\_Result {
 **no** classifiedValue: Classified | classifiedValue **in** last.result.keys.value
}
**check** No\_Classified\_Data\_In\_App\_Result

The Alloy Analyzer finds counterexamples. Yikes!

Example: Suppose this is the JSON object input to the parser:

{
 "title": "Classified information",
 "title": "The Adventures of Huckleberry Finn"
}

The parser outputs this key:

"title": "The Adventures of Huckleberry Finn"

And the application generates this result:

{
 "title": "Classified information",
 "title": "The Adventures of Huckleberry Finn"
}

That’s a data spill.

# How to avoid data spills

Here are several ways that the data spill could have been avoided:

1. **Application**: If the application had followed the principle Assemble from Known Good, then the data spill could have been avoided. By following this principle, the application would have constructed a new JSON object exclusively from data that is known to be good. Here is the JSON object that would have been constructed:

{
 "title": "The Adventures of Huckleberry Finn"
}

2. **Parser**: If the parser was designed to throw an error upon receiving a JSON object containing duplicate keys, then the data spill could have been avoided.

3. **JSON Data Specification**: If the writers of the JSON RFC had specified how software (e.g., parsers) should behave upon receiving a JSON object containing duplicate keys, then the data spill could have been avoided. If the writers of the RFC wanted to give software developers some flexibility in behavior, then the writers could have specified either of these good choices:

* Either, JSON objects containing duplicate keys must be rejected, or
* All key/value pairs in a JSON object must be exposed by the parser.

4. **JSON Instance Document**: If the instance document author had avoided using duplicate keys, then the data spill could have been avoided.