

# Security analysis of JSON parser

Ravibabu Matta, Martin Ukrop, Jiří Weiser

December 9, 2014

## 1 Introduction

We analyzed a JSON parser written in C++ by Vít Šesták and Daniil Leksin. According to the specification, the parser is slightly more permissive than it should, some examples are provided. Nevertheless, the authors claim that no inputs should lead to memory corruption. The specification assumes a valid UTF-8 stream on the input.

## 2 Static and dynamic analysis

Microsoft *PreFast* reports no issues and *CppCheck* has found just a single (but meaningful) style issue: exceptions should be caught by reference, not by value. *Valgrind*'s memcheck tool reports no memory issues on multiple JSON inputs tested.

## 3 Manual code inspection

Manual testing and code inspection revealed the following issues:

- The parser is able to parse numbers which are not allowed by JSON standard, such as `-.1`. However, such inputs cause no memory corruption or parsing problems.
- The parser should check the end-of-file character after the last token. Omitting this check allows for multiple subsequent JSON strings to be parsed. However, such inputs cause no memory corruption or parsing problems.
- The parser is not able to handle unfinished strings correctly. For instance, the input string `"` (single double-quote character) causes the parser to crash with segmentation fault.
- The parser accepts some string characters inside the range `0x00–0x1F` which should be refused. These characters should be encoded in Unicode escape sequence.
- The parser is not capable of accepting valid Unicode escape sequences of characters larger than `0xFFFF` (e.g. `"\uD852\uDF62"`). However, such inputs cause no memory corruption or parsing problems.
- The parser is not capable of accepting valid Unicode escape sequence of some characters less than `0xFFFF`. For example, using the Euro sign on the input (`"\u20ac"`) causes a problem in the code. The stack is not exposed, only UTF-8 characters higher than 127 are badly flushed.
- The `Json::Value` object does not define a virtual destructor. Even though this does not lead to a memory leak in this case, that is only due to a (somewhat improper) usage of `std::shared_ptr`.

## 4 Conclusions

The automated analysis did not find any major flaws. Manual inspection revealed several issues, bringing into attention the following three major problems:

- The function `shared_ptr<Json::String> Json::String::readStringFrom(istream &in)` does not return any value if the ending quotation mark is not reached. This results into uninitialized `std::shared_ptr< Json::String >` and therefore a segmentation fault.
- The function `void Json::String::parseHexaStringSequence(istream &in, stringstream &ss)` does not accept some Unicode escape sequences less than `0xFFFF`. Such characters are flushed incorrectly.
- The `Json::Value` object should define a virtual destructor.