

Code review of S-expressions parser

Team G (David Formánek)

December 8, 2014

1 Reviewed software

S-expressions are data structures for representing arbitrary complex data. Each expression is either byte string or a list of simpler S-expressions. The aim of the reviewed software is to parse these expressions and output their canonical form or print an error (for invalid expressions). Parser is implemented in C++ language using two custom classes (*Parser* and *CanonicalWriter*) plus a structure representing an S-expression.

2 Process of analysis

Analysis has started with study of the S-expressions specification and manual dynamic testing of various inputs. The *Valgrind memcheck* tool has been used to detect memory leaks. The *MSVC* compiler set to warning level 4 has shown warnings at 14 locations, *gcc* compilation (with flags `-Wall -Wextra -pedantic`) has put warning at 2 locations (identical to 2 warnings from MSVC). Static analysers *PREFast* and *Cppcheck* has found no errors. The whole code has been statically reviewed manually utilizing results from previous analysis.

3 Found problems

Valid tokens starting with colon are not accepted

According to specification, tokens may begin with punctuation (including `:`, e.g. `:token`). However, the reviewed software considers such tokens to be a verbatim encoding with missing length and therefore rejects them. The problem in code is at line 182 in file `sexpressions.cpp`, instead of throwing an exception, `_parse_token` function should be called.

Missing closing quote causes infinite looping

When processing of a quoted string starts, next characters are read until quotation mark is closed. If the value for the end of file is read, the reading loop still repeats (e.g. for input `"repeat`). Not only this fully utilizes a processor core but the value of end of file is added as data to S-expressions structure and all memory is consumed in a while. Exploiting this could cause an effective denial-of-service attack. To solve the issue, add a check for end of file inside the loop starting at line 232 (`sexpressions.cpp`).

Invalid verbatim encoding is accepted if file ends

For verbatim encoding, as many bytes is read as is the specified length. If the specified length is greater than the real one and file ends (e.g. for input `10:short`), assumed reading continues and following length check is useless. As a result, input is accepted with non-existing data created as return values for end of file converted to *unsigned char* type (bytes with value 255). Add end of file detection inside the loop starting at line 218 (`sexpressions.cpp`) to solve the problem.

Program can crash while reading long valid expressions

After successful parsing, data are output using iteration over child expressions and over bytes of octet-strings. For these iterations, signed *int* type is used as a loop variable while sizes are represented by unsigned *size_type* with greater maximum possible value. Consequently, if there is an octet-string with length greater than *INT.MAX* (or great count of child expressions), loop variable overflows and array is accessed using negative index. This behaviour can be prevented by using *size_type* for loop variables at lines 516 and 534 in `sexpressions.cpp`. The issue has been found by both MSVC and gcc compilers.

Sub-expressions are never deallocated

After reading character (, *parse_expression* function creating new instances of *Expression* structure is called for each sub-expression. These structures are never deallocated regardless the input is valid or not. Proper destructor (with recursive deallocation) could be implemented to prevent memory leaks.

More memory leaks if expressions are invalid

Exceptions are thrown to indicate an invalid input. These are caught in main function, error message is printed and no deallocation is performed before the value of 1 is returned so structure with invalid expression and class instances are leaked. Moreover, exceptions are allocated on the heap without possibility to deallocate them and this causes another memory leak. To solve this problem, add deallocation before function return and do not use operator *new* when throwing exceptions.

4 Conclusion

Reviewed parser contains several serious issues – there are both valid inputs that are rejected and invalid inputs that are wrongly accepted, both valid and invalid inputs causing program not to end in proper way (crashing or infinite looping) and the most of memory is not deallocated. General recommendation is to always check the end of file, study specification carefully, do not ignore compiler warnings and ensure the memory is deallocated for every possible program flow. Problems have been found mostly by manual static and dynamic analysis but outputs from MSVC compiler and Valgrind have been quite helpful.