

TERM PROJECT REPORT

PA 193- SECURE CODING PRINCIPLES

Submitted by

Archit Agarwal (UCO-436264)

Karel Kubíček (UCO-408351)

Umang Kumar (UCO-430491)

PART -1

This part consisted of checking the code using Static checker.

1. **Pre Assessment Review:-** Our team has chosen the source code of the text editor – “geany”. Geany is a small and lightweight Integrated Development Environment. Source Code is of around 65K lines.

2. **Application Review:-** It was developed to provide a small and fast IDE, which has only a few dependencies from other packages(GTK). Main features of Geany are as follows

- (a) Syntax highlighting
- (b) Code folding
- (c) Symbol name auto-completion
- (d) Construct completion/snippets
- (e) Auto-closing of XML and HTML tags
- (f) Call tips
- (g) Many supported file types including C, Java, PHP, HTML, Python, Perl, Pascal Symbol lists
- (h) Code navigation
- (i) Build system to compile and execute your code
- (j) Simple project management
- (k) Plugin interface

3. **Analysis:-** Geany is known to run under Linux, *BSD, MacOS X, Solaris and Windows. More generally, it should run on every platform, which is supported by the GTK libraries. Only the Windows port of Geany is missing some features. The code is licensed under the terms of the GNU General Public Licence.

4. The source code of geany was downloaded from <https://github.com/geany/geany>, works both on linux and windows.

5. The following methods/checkers were used to check the code:-

- (a) Manual Code Review
- (b) Static Checker- CPP Check
- (c) Dynamic Checker- Valgrind and its associated subtools.

6. Prefast could not be used as there were external dependencies in the source code which are not supported by MSVC. This was discussed with Zdenek Riha.

7. **Testing Procedure:-** The source code of the ‘geany’ was downloaded from github. The tools as mentioned above were downloaded, installed and configured from respective sources and were used to check the source code.

8. Results and Observations:- During the checks different types of errors/ warnings were observed in the source code. The summary of these are tabulated as below:-

- (a) CPPcheck:- Using CPPcheck, we found 75 style warnings (mainly that scope of variable can be reduced or some return statement is unnecessary), and 5 warnings about possible null pointer dereference. 31 of style warnings is false positive, others can be changed. We have corrected all 46 correct warnings, so we can send them to geany team. 4 of 5 null pointer dereference warnings are false positive. But this one can actually happen: [filetypes.c:1273] -> [filetypes.c:1256]: (warning) Possible null pointer dereference: ft - otherwise it is redundant to check it against null.: in case that function document_get_current returns NULL (which is possible), than this is true. We suggest to add check for NULL pointer here.
- (b) Valgrind tools:- Usage of Valgrind tools is difficult, because most of memory leaks and parallel problems, are created from GTK. But developers of GTK says, leaks are part of optimization. For testing GTK application with Valgrind, GTK suggest to use GTK suppression file. Using them reduced the number of leaks by 2/3, but it still means, that output is still full of GTK problems. No leak or thread problem caused by geany was found.
- (c) Manual check:- We tried to compile the source code using provided make file, so we can see warnings from compiler (gcc). This make file disables the warnings. It was not possible for us to change the make file to get warnings due to complicated nature of make file. Another part was manual reading of source code. We looked for TODO and FIXME in code comments (there are 62 FIXME and 38 TODO in the whole code). Lot of these comments are due to backward compatibility. Some of them are really hacks and few are programming mistakes. One of them can be fixed this way:

keyfile.c from line 105:

```
static struct
{
    gint number_ft_menu_items; //should be guint
    gint number_non_ft_menu_items; //should be guint
    gint number_exec_menu_items; //should be guint
} build_menu_prefs;
```

If gint is changed to guint, than in build.c line 2768 FIXME will be solved. There should be more rewriting, but it is possible and such corrections can reduce count of warnings during compilations.

9. Remediation Action and Support:- All the errors were reviewed one by one, and attempts were made to rectify the code to remove these errors. Total 46 errors were rectified by our team as mentioned above. Our team is planning to forward the rectified code along with the description of all the errors to the development team of 'geany' for their comments/ action.

PART-2

1. This part was to write the code for implementing a parser. Our team has implemented the parser which is a subset of Doxygen. This parser specifically parses the javadoc style documents. Brief details of the implementation and its features and limitations are mentioned in the succeeding paragraphs.

Brief of Doxygen

2. Doxygen Official site is <http://www.stack.nl/~dimitri/doxygen/manual/index.html>. Since Doxygen supports the standard Javadoc tags we can run Doxygen on any source code with Javadoc comments on it.

Details of Implemented Parser

3. Our implementation is in C++ 11, following are the features of the implemented parser:-

- (a) For starting doxygen keyword, '@' character has to be used.
 - (b) Each file should begin with doxygen comment, keywords @author and @file (usually for C/C++) or @version (Javadoc style) are necessary.
 - (c) Each doxygen comment have to specify some function, if there is no function below, then warning message is obtained.
 - (d) Comment for function header - @brief with basic information about function, @param for parameters of the function and @return for specification of returned value are suggested.
 - (e) Header comment for file - @author is mandatory, @file in C/C++ or @version for Java is also mandatory, @see, @link and @since is allowed.
-

How to Use the Parser

4. On execution of the code input file containing the code (document to be parsed) is to be provided as command line argument (ensure correct path is provided for the input file). Other files are main.cpp, parser.cpp, tokenizer.cpp and their header files(for parser.cpp and tokenizer.cpp). One can use MSVC 13 for this. Create new project and add the above mentioned files and compile and execute. The output will be displayed on the standard output device.

Known Issue/ Limitations

- (a) Do not document overloaded operator (), it will be taken as function with no arguments.
 - (b) Do not document classes, structures and enums (not supported yet).
 - (c) @ in mail address is interpreted as keyword - just warning
 - (d) Path to the @file do not support in windows using \ (recommendation: input file to check in same directory as binary and as give it to the program by relative address (just input.c)
-

PART-3

Part 3 of the project comprised of review the code of the implemented parser by another team.

1. **Pre assessment:-** Our team's task was to review the code of team B with the objective of finding bugs and issues. Team B has implemented the subset of XML parser. The code consisted of around 900 lines in 3 files.

2. **Application Review:-** Rules for the parser are small subset of XML specifications. The scope of the project along with specification and the rules are provided in the document.

3. **Analysis:-** The following checkers were used to check the code:-

- (a) Manual Code Review
- (b) Static Checker- CPP Check
- (c) Dynamic Checker- Valgrind and fuzzer filep

4. Results and Observations

(a) **CPP check:-**

Only one warning was observed. Code was tested by author, so not many warnings.

(b) **Prefast:-**

No bugs found.

(c) **Manual Review – notes on coding style:-**

- i. Constructors are used; whereas static factory will be a better choice.
- ii. If without else are used.
- iii. Redundant return statement are used (line 33)
- iv. Multiple type of comments.
- v. Readability of code is not good. Confirmed by tool Source Monitor also with score 28.

(d) **Manual Review – running code on various inputs:-**

As the standard of XML was specified by authors, the main part of searching for bugs (for example searching for wrongly accepted or rejected inputs), was searching for mistakes in specifications. The parser is written well for the given specification, so there were not bugs in it. However our team has discovered following issues:-

- (i) The parser rejects empty elements, which has space between '/' and '>'. For example `<a / >` is rejected with error code 10244, which means `PARSER_BAD_NODE`, `PARSER_MISMATCHED_TAGS`, `PARSER_MISMATCHED_TAGS`, `PARSER_ILLEGAL_CHAR`, `PARSER_ILLEGAL_CHAR`. This input is not forbidden by specifications, so it should be valid.
- (ii) Function `parse_node` is written in recursive form (which is called from infinite cycle `for(;;)`), we suggest using `while(notFinished)` for better readability). It is recursively called for each

nested element. In case of big file with long branches of nested elements (<40k of nested elements), the parser fails on stack overflow. We suggest to rewrite recursion to iterative form. Alternative is counting depth of recursion with limit set to 1000, which is usable for all usual files. Also it takes long time for large files, which can lead to DoS attack, if this parser is used on server.

(e) **Dynamic analysis tools:-**

If the input file is not provided at cmd prompt the valgrind is showing 1.5 m events recorded.

Also for large files (over 100kB of input), parser crashes with Segmentation fault and Valgrind outputs large number of incorrect freed memory.

Usage of fuzzer filep showed, that parser is not vulnerable to little corrupted inputs. The specifications are so strict, that little change of syntax markers leads to input rejecting.

Masiff:- Sometimes we found that heap is not empty in some cases.

(f) **Test Cases:-**

Various types of XML files were parsed using the parser. Most of the files were successfully parsed considering the limitations as mentioned in the document provided.

(g) **Other suggestions:-**

The code is written mainly as library, which is able to parse given XML format. For example Parser class has only 2 public methods (except ctors and dtors): parse and get_error_code. This is well done secure API, but the effort on creating parser program was lower. The output of program is parsed document or error code. I suggest to create at least error_to_string function, which produces human readable error from error code, and also position, where the error is located, would be helpful.

Only suggestion about secure API is using enum class instead of enum, because enum is globally accessible, however enum class is closer to Java enums – it is just part of written class. This functionality is only available in C++11, but it is used in this project.

5. **Conclusion**

Parser is well written library, which can be used for parsing specified XML format. The automated analysis did not find any major flaws. These 3 mainly problems were found manually:

- Code readability is not good. It may lead to unmaintainable code. Many functions can be split into shorter subfunctions, whose can be reused.
- At least one mistake in specifications was found, there may be others.
- The parser is not prepared for large inputs. This may lead to memory corruption and revealing some secret information to attacker. In case of using on some server, complexity of parsing files can lead to DoS attack for parsing large inputs.