STATIC CODE ANALYSIS

Alexandru G. Bardas[1]

Abstract

A lot of the defects that are present in a program are not visible to the compiler. Static code analysis is a way to find bugs and reduce the defects in a software application. This paper gives you an overview on static code analysis, well-known tools and the benefits of this practice.

Introduction

In the 1970's, Stephan Johnson, then at Bell Laboratories, wrote Lint, a tool to examine C source programs that had compiled without errors and to find bugs that had escaped detection.
There are many ways to detect and reduce the number of bugs in a program. For instance in Java, *JUnit* is a very useful tool for writing tests. Overtime research proved that analyzing the code (especially code reviews) is the best way to eliminate bugs. This is not always possible because it is very hard to train people and get them together to study and identify problems in programs. Furthermore it is almost impossible to use code inspections on project's complete code base.
Most errors fall into known categories, as people tend to fall into the same traps repeatedly. Therefore a static analyzer or checker is a program written to analyze other programs for flaws. These type of programs are scanning the source code (see Figure 1), the byte code or the binaries of a program in order to match patterns.
Static analyzers have the potential to find rare occurrences or hidden back doors. Since they consider the code independently of any particular execution, they can enumerate all possible interactions[4] between the different components or modules.

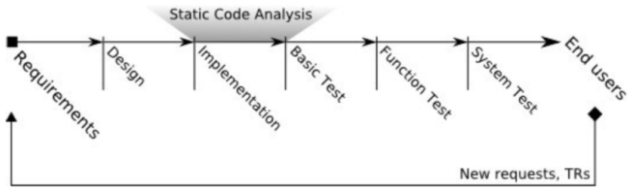

Figure 1 The intended usage of static code analysis in an example development cycle

Why Start With Static Source Code Analyzers?

Higher level representations, such as requirements or use cases, are better places to prevent flaws. So far these areas are not mature enough for standardization.
Roughly half of all security weaknesses are introduced during coding, so making improvements after high level design may be very helpful.
Unlike binary or byte code, source code can be read by humans. Also there are many tools that work with source code. For these reasons, source code seems a good place to begin.

[1] Phd Student, Kansas State University, United States of America

What should a source code analyzer do?

A source code analyzer should find weaknesses and report their severity and location. The weakness class corresponds to Common Weakness Enumeration (CWE) entries. There are a lot of tools that also report conditions that may expose the weakness, data or control flow related to it, more information about that class of weakness including examples of how to fix it, the certainty that the weakness is a vulnerability (not a false alarm), or some rating of the severity or ease of exploit.

Optionally a tool can construct a report that could be used by other tools. In order to be practical in repeated runs, the source code checker must have some mechanism to suppress reports of weaknesses judged to be false alarms or otherwise to be subsequently ignored.

False positives are a critical factor in static code analysis. Theoretically, static analysis tools compute a model of a program that they analyze for certain properties. Since static analysis problems are generally undecidable, “either the computed model is approximate or the analysis is approximate”. Because of these approximations, tools may miss weaknesses (false negatives) or report correct code as having a weakness (false positives). To be recognized and adopted a tool must "have an acceptably low false positive rate".

Types of Static Code Checkers

Static code analyzers come in different flavors, analyzers that work directly on the program source code and analyzers that work on the compiled byte code. Each type has its own advantages. When analyzing directly the program code, the source code static analyzer checks directly the source program code written by the programmer.

Compilers optimize code, and the resulting byte code might not mirror the source.
On the other hand, working on byte code is much faster, which is very important on projects that contain for instance more than one hundred thousand lines of code.

Even though each code checker works differently, most of them share some basic traits. Static checkers read the program and build an abstract representation of it that they are using for matching the error patterns they recognize. On the other hand static checkers perform some kind of data-flow analysis, trying to infer the possible values that variables might have at certain points in the program. When a program accepts input, there is a possibility that this input can be used to subvert the system. Buffer overflows have been over time hacker’s favorite inroad. Nowadays SQL injection seems to have taken the top place for program sore spots. Therefore data-flow analysis is very important for vulnerability checking. It is essential to be able to trace the input flow from users through the program.

There is no code checker that can ever assure developers that a program is correct. Such guarantees aren’t possible.

“In fact, no code checker is complete or sound. A complete code checker would find all errors, while a sound code checker would report only real errors and no false positives.”

The percentage of false to true positives indicates if a code checker is suitable for different programs. It is recommended for developers to examine a checker's behavior in their work before committing to it in the whole project.

Human fallibility is somewhat predictable, but code checkers cannot take all possible bugs into account. Most of the checkers gives programmers the option to define their own rules for the checker to use. In this way, if developers are certain or can predict that they are particularly prone to some kinds of bugs, they can guard against them by writing custom bug detectors.

Eliminating bugs doesn't ensure high program quality. Quality metrics can be used to do that.

Strengths and Limitations

Every static analyzer has a database of vulnerabilities to look for in code. Most of the products have the option of adding custom rules. Static analysis may be performed on modules or unfinished code, although the more complete the code, the more thorough and accurate the analysis can be.

On the other hand testing requires test cases or input data. Testing also requires artifacts that are complete enough to be executable, possibly with supporting drivers, stubs, or simulated components.

These are the reasons why sometimes it is more convenient to us static code checkers during the development process. Static code analysis does not replace testing, it should complement it. Therefore static code checkers should never replace testing because testing has the advantage of possibly revealing completely unexpected failures.

Static analyzers have the potential to find rare occurrences or hidden back doors. Static checkers consider code independently of any particular execution, they are analyzing the code without compiling it. In this way static checkers can enumerate all possible interactions between the different modules and components of the analyzed code.

The number of interactions tends to increase exponentially, defying comprehensive static analysis and test execution alike. Static analysis can focus on the interaction without testing's need to re-establish initial conditions or artificially constrain the system to produce the desired interaction.

For instance it is impossible to expect that black-box testing can discover a backdoor accessible when the username is "AlexBardas" since there are a nearly infinite number of arbitrary strings to test.

Analyzers are limited by the sophistication of the reasoning in them. A good example in this sense are static source code analyzers that do not handle function pointers and few can deal with embedded assembler code.

Even in cases when the models of the programming language, compiler, hardware, and other pieces used in execution are perfect, analyzers have the same fundamental limitation as any other logical system:

They cannot solve the halting problem or undecidable problems.

This does not need to be a serious limitation in practice. Important code should be so clearly correct that it confuses neither human nor tools.

Even though running tests is straightforward, it can be sometimes challenging to develop tests that exercise a particular property or module. When new attacks or failure modes are discovered, new tests must be developed.

In case of static analyzers the vulnerability database has to be updated as well when new vulnerabilities are discovered.

A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation.

If the set of vulnerabilities is not updated then the static checker will miss the new weakness in the analyzed code. The advantage of a static analyzer is that once the vulnerability is added to the set and validated, then the analyzer can be rerun on all code. On the other hand test generators can give a similar advantage for generating test cases to exercise the latest vulnerabilities.

Static analysis is no panacea. There are subtle and complex vulnerabilities that can always defeat the reasoning in a static analyzer. For instance the lack of auditing or encryption cannot reasonably be deduced from only the examination of post-production artifacts. Software with no resiliency or self-monitoring is open to errors in installation or operation, but static analysis can be one of the last lines of defense against vulnerabilities.[4]

How can we construct good software?

The qualities needed in today's software and systems cannot be "tested in".
Characteristics, such as security, safety, and reliability, must be designed in and built in from the beginning. The development process must be constructed such that users can rely upon the resulting software.

Even when implementing and using the most disciplined and well-characterized processes, artifacts must be analyzed to gain assurance that the output of the process is close to the target qualities. This is the main characteristic of quality control.

Static Analysis Tools

Usually, like almost in any other software application, there are two flavors of static analysis tools: open-source and commercial static code analysis tools. It depends on the developers and the company what type of product they use.

Some examples of well-known and widely used static code checkers are Coverity Static Analysis, Fortify, FindBugs and Splint / LCLint (Lint - static source code analysis tool).

Coverity Static Analysis

Coverity Static Analysis is a commercial product and it pretends to be the leading automated approach for ensuring the highest-quality. Coverity Static Analysis automatically scans C/C++, Java and C# code bases with no changes to the code or build system.

Because it produces a complete understanding of your build environment and source code, Coverity Static Analysis is the tool of choice for developers who need flexible, deep, and accurate source code analysis.
Benefits of Coverity Static Analysis:

- o Automatically find critical defects that can cause data corruption and application failures[18]
- o Improve development team efficiency and speed time to market for critical applications[18]
- o Improve software integrity and end-user satisfaction

Coverity Static Analysis works in three steps to achieve software integrity:

1. Map the Software DNA
2. Indentify Critical Defects
3. Resolve Defects

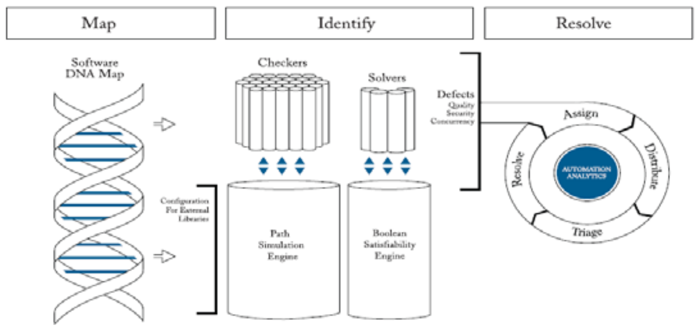

Figure 2 Coverity's three-step process provides a comprehensive methodology for proactively finding and eliminating critical software defects.

Fortify

Fortify 360 is also a commercial product that identifies, prioritizes and helps developers eliminate security vulnerabilities in software. It delivers:

- o **Vulnerability Detection (**detect vulnerabilities with static and dynamic analysis)
- o **Collaborative Remediation** (fix vulnerabilities in a shared workspace **Reporting and Governance** Manage and report on the process)
- o **Threat Intelligence** (stay ahead with cutting edge research)

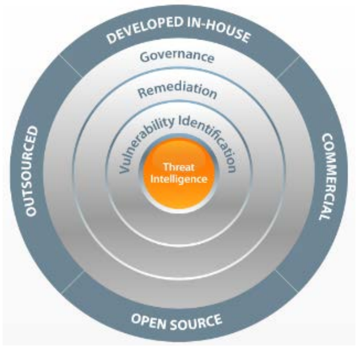

Figure 3 Fortify options visualization

FindBugs

FindBugs is a program which uses static analysis to look for bugs in Java code. It is free software, distributed under the terms of the Lesser GNU Public License. The name FindBugs™ and the FindBugs logo are trademarked by The University of Maryland. As of July, 2008, FindBugs has been downloaded more than 700,000 times.

FindBugs requires JRE (or JDK) 1.5.0 or later to run. However, it can analyze programs compiled for any version of Java.

Splint / LCLint

Splint is an open-source tool that runs mainly on UNIX based operating systems. This tool is used for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint.

Do Static Code Analysis Tools Really Help?

Developers can think of several potential problems with the use of static code analysis tools in practice.

A tool may report many unimportant weaknesses, but miss the small number of serious weaknesses that really affect security.
If a developer takes a mechanical approach to fixing weaknesses reported by tools, programmers may not think as much about the program logic and miss more serious vulnerabilities.

On the other hand, the developer may spend time correcting unimportant weaknesses reported, making other mistakes in the process and not having as much time for harder security challenges.

Recognizing this kind of problems, Dawson Engler articulated the question: "Do static source code analysis tools really help?"

Coverity was funded by the US Department of Homeland Security. In collaboration with Stanford University, they analyzed over 50 open-source projects since March 2006.
For example Coverity scanned MySQL version 4.1.8 in early 2005. Version 4.1.10, released 15 February 2005, contained fixes based on Coverity reports. Figure 4 compares the vulnerabilities discovered in version 4.1.10 or later versions with vulnerabilities discovered before the 15 February release.

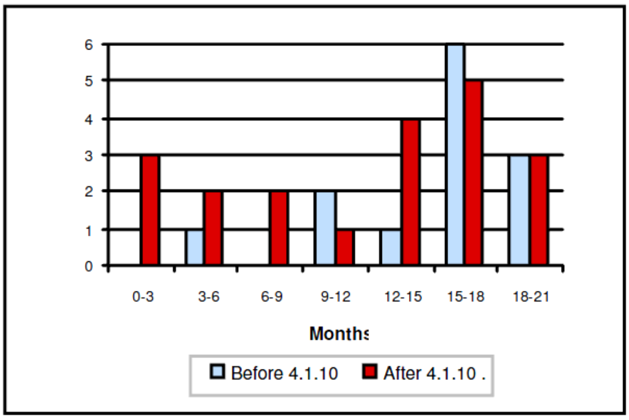

Figure 4 MySQL vulnerabilities before and after 4.1.10 [8]

The Red bars, on the right, are vulnerabilities discovered in version 4.1.10 or later. These bars are grouped by the discovery date reported in the National Vulnerability Database.

The data covers 21 months after the release of version 4.1.10. The light blue bars, on the left, describe vulnerabilities discovered before the release. The process began 21 months before the release that is May 2003. Vulnerabilities discovered after 15 February 2005 that were only present in versions before 4.1.10 were not taken into account. This data is insufficient to draw final conclusions.

On the other hand, Dejan Baca, Bengt Carlsson and Lars Lundberg presented in their paper named *Evaluating the Cost Reduction of Static Code Analysis for Software Security* a case study in which mature software with known vulnerabilities is subjected to a static analysis tool. The three authors believe that the faults/bugs constantly introduced into programs during their development can be reduced by using static analysis tools.

Faults and flaws can propagate during runtime into failures that might be exploited as vulnerabilities, see Figure 5.

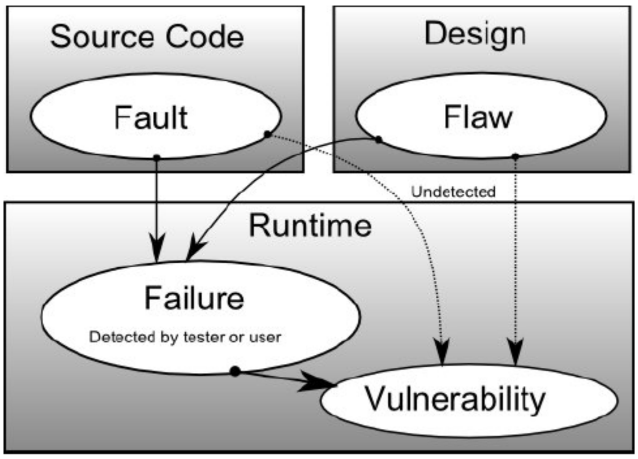

Figure 5 The relation between source code faults and design flaws resulting in visible failures that might propagate into known or unknown vulnerabilities. [17]

A trouble report or bug report (TR) includes any bugs that are found by the users are then reported back.

After looking at real-life trouble reports from three large software systems, consisting of approximately 1,000,000 lines of C++ code, Baca, Carlsson and Lundberg noticed a significant cost associated with handling the security related trouble reports in these systems.

The research study showed that, by using a static analysis tool a 17% cost reduction for reported security bugs would have been possible. One product even showed a 23% cost reduction. The cost reduction includes all costs associated with the static code analysis tools. Most of the vulnerabilities found in the code were stack or heap related, i.e. the security detection capabilities mostly rely in the software's handling of memory. The authors also stated that no design based vulnerabilities were detected and more implementation failures should have been possible to detect, i.e. when the static analysis tool is lowering the rate of false positives some vulnerabilities are dismissed.

Another interesting aspect is the fact that almost 70% of the trouble reports were not found by Coverity, i.e. for these there were no reduction of costs. Product A had the best effectiveness with 37.5% of the trouble reports detected while Product C had the worse with 25%, but because the static code analysis tool also found dormant vulnerabilities that were not reported as trouble reports. A total of 2.6 times more vulnerabilities were detected by the static code checker compared to the trouble reports. This means that static code analysis does not only reduce the cost. There is also a significant quality improvement due to the detection of dormant vulnerabilities.

Conclusion

Programming is one of the toughest jobs in project development. No machine can substitute for good sense, a solid knowledge of fundamentals, clear thinking, and discipline, but bug detection tools can help developers.

Static analyzers should be a key part of every software development process.

References

1. S. Johnson, *Lint: A C Program Checker*, tech. report 65, Bell Laboratories, Dec.1977.
2. P. Louridas, “JUnit: Unit Testing and Coding in Tandem,” *IEEE Software*, vol. 22, no. 4, 2005, pp. 12–15.
3. P. Louridas, “Static Code Analysis”, IEEE Software JULY/AUGUST 2006 (Vol. 23, No. 4) pp. 58-61
4. Black, Paul E. "Static Analyzers in Software Engineering", March/April 2009
5. Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction.", 2008
6. Chelf, Ben, and Christof Ebert. "Ensuring the Integrity of Embedded Software with Static Code Analysis.", May/June 2009
7. Black, Paul E. "SAMATE and Evaluating Static Analysis Tools.", September 2007
8. Black, Paul E. "Software Assurance with SAMATE Reference Dataset, Tool Standards, and Studies.", October 2007
9. Chinchani, Ramkumar, and Eric Van Den Berg. "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows.", 2005
10. Bush, William R., Jonathan D. Pincus, and David J. Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors.", December 1999
11. Bergeron, J., M. Debbabi, J. Desharnais, M.M Erhioui, Y. Lavoie, and N. Tawbi. "Static Detection of Malicious Code in Executable Programs.", 2001
12. Holzmann, Gerard J. “Conquering Complexity.” Computer 40 (12): 111-113, Dec. 2007.
13. “Source Code Security Analysis Tool Functional Specification Version 1.0.” National Institute of Standards and Technology (NIST), Special Publication 500-268. May 2007 <http:// samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268. pdf>.
14. Gary McGraw (2006), *Software Security*, Addison-Wesley.
15. Source Code Security Analysis Tool Functional Specification Version 1.0, National Institute of Standards and Technology, Special Publication 500- 268, May 2007. Available at http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf (Accessed 24 May 2007).
16. *Accelerating Open Source Quality*, http://scan.coverity.com/ (Accessed 21 May 2007).
17. Baca, Dejan, Bengt Carlsson, and Lars Lundberg. "Evaluating the Cost Reduction of Static Code Analysis for Software Security.", May 2008
18. Coverity Website – http://www.coverity.com/library/pdf/CoverityStaticAnalysis.pdf
19. Fortify Website – http://www.fortify.com/products/
20. Findbugs Website –http://findbugs.sourceforge.net/
21. Splint Official Website – http://www.splint.org/