

ORACLE SOLARIS STUDIO CODE ANALYZER

KEY FEATURES

- Integrated and comprehensive view of common coding errors via easy-to-use GUI
- Accurate analysis that is faster than competitive alternatives
- Low false positive rate
- Static code checking that detects common programming errors as part of normal build process
- Dynamic code checking that finds memory-related errors in executed code paths
- Code coverage checking that informs developers about gaps in test coverage

BENEFITS

- Improves software quality, security, and reliability
- Increases developer productivity

The Oracle Solaris Studio Code Analyzer ensures application reliability by detecting application vulnerabilities, including memory leaks and memory access violations, enabling developers to write better code with fewer errors faster.

Introduction

Have you ever been called in the middle of the night because your application crashed? Does your application exhibit mysterious intermittent failures that are hard to pinpoint? Do you think your software is not adequately tested? The Code Analyzer helps identify application reliability and security issues by utilizing dynamic, static, and code coverage analysis to detect 45+ common coding errors, including memory leaks and memory access violations faster than competitive alternatives.

The Code Analyzer performs static analysis when you are compiling your application, and it performs dynamic analysis when you are running your application and gives you feedback about where you may have errors. In addition, it provides code coverage data to give you information about functions that are not covered by your test suite and provides guidance on the type of benefit you could get by covering those functions. The Code Analyzer provides a comprehensive view of application vulnerabilities by synthesizing the data collected from these three types of analysis, enabling you to improve application correctness and reliability. It also provides advanced error filtering and sorting capabilities, enabling you to track, detect, and fix issues faster.

Static Analysis

Modern static checking can uncover implementation defects in software earlier, more reliably and at a far lower cost than conventional testing methods. Unlike early UNIX tools such as `lint`, programs can now be analyzed at a semantic level to point out real defects, rather than “potentially problematic constructs”. Using sophisticated analysis techniques, bugs and implementation defects can be found during compilation and fixed right away, saving enormous amounts of time and resources.

Static analysis is enabled in the compiler when building your application. Some of the useful errors found during this phase include:

- Reading and writing beyond array bounds
- Incorrect `malloc` and freed/freeing memory issues
- Null pointer dereference (leaky pointer checks)
- Infinite empty loop

- Uninitialized memory reads/operations
- Type cast violations

All of these types of errors, and many more, are detected during regular builds. Other than the addition of a special option, no other change is necessary. Developers typically find that detecting and eliminating these errors during the design and early implementation phase is an order of magnitude cheaper than detecting them later during development or having to generate patches for critical bugs.

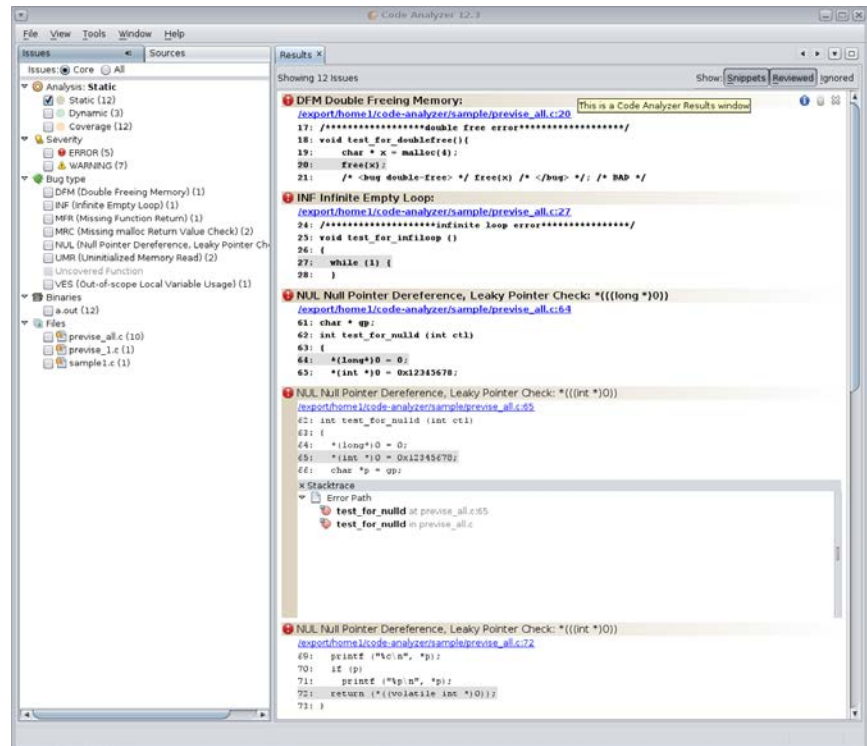


Figure 1: Comprehensive view of static errors

However, not all errors are detectable at compile time. Some real errors may not be reported (these are called false negatives) and some reported errors might not actually be issues (these are called false positives). The goal of the tool is to minimize these types of errors. Another real limitation is that some errors depend on data that is available only at runtime. For such errors, the tool offers an additional dynamic code checking facility. The advantage of using the same compiler to produce static errors is two-fold: What is compiled is exactly what is checked and the tool does not use any other external parsing technique which may or may not analyze the same code the compilers see during build time.

Dynamic Analysis

While static code checking is extremely useful, it does have some limitations outlined above. Additionally, developers want to know exactly how an issue arose during application runtime. Dynamic checking provides a complementary view in discovering common kinds of errors:

- Reading from and writing to unallocated memory
- Accessing memory beyond allocated array bounds
- Incorrect use of freed memory
- Freeing the wrong memory blocks
- Uninitialized Memory reads / writes
- Memory leaks

Dynamic checking works on binaries built with the Oracle Solaris Studio compilers. No special compilation flag is necessary, although the presence of `-g` is recommended to help identify offending source lines. Access to source code is not required, which means it can be used on production binaries and it works well with third-party libraries. The binary is instrumented for these memory related errors. Due to the close linkage with the compilers and intimate knowledge of hardware and Oracle Solaris interfaces, the overhead incurred during dynamic checking is the smallest among similar tools.

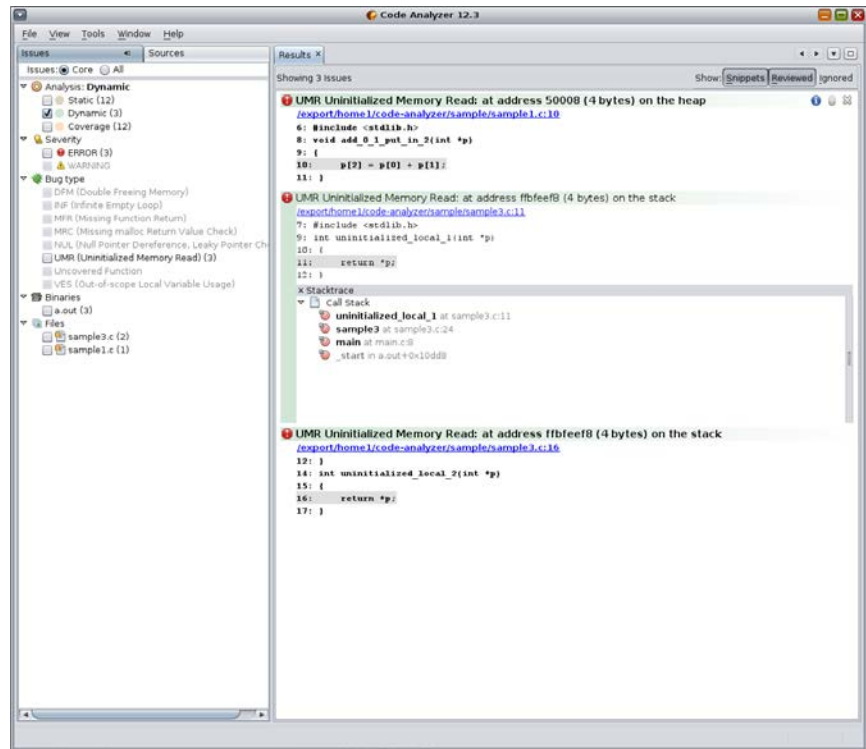


Figure 2: Comprehensive view of dynamic errors

Code Coverage to Identify Gaps in Testing

Code coverage checking uses binary instrumentation to inspect test suite runs of an application and to identify vulnerabilities by highlighting source fragments that are not covered. Highlights include the following:

- Ability to collect and aggregate uncoverage data over multiple runs, thus making it suitable for integration during automated product testing
- Ability to display potential coverage percentage
- Multithreaded and multiprocess safe
- Low overhead of instrumentation during runs

The code coverage checking feature of the Code Analyzer provides a sorted list of most important functions that have not been tested. These are functions with the largest functionalities. It also hides uncovered functions that are subsumed by other functions, reducing clutter.

Easy-to-Use Graphical Interface

Based on the award-winning NetBeans framework, the Code Analyzer GUI provides an easy-to-use graphical view of the data collected by these three types of analysis.

The tool opens with two panes: One pane highlights the types of vulnerabilities found and the other pane details the errors in the context of the surrounding code fragments to quickly pinpoint the root cause of the issues. The feature-rich GUI provides the following:

- Filters to enable focus on select types of vulnerabilities or a selection of source files to focus on
- Buttons to hide, show, and mark vulnerabilities according to the user's preference
- Integrated Editor for easy source fixes
- Source browsing: class, method, field usages, and call-graph information

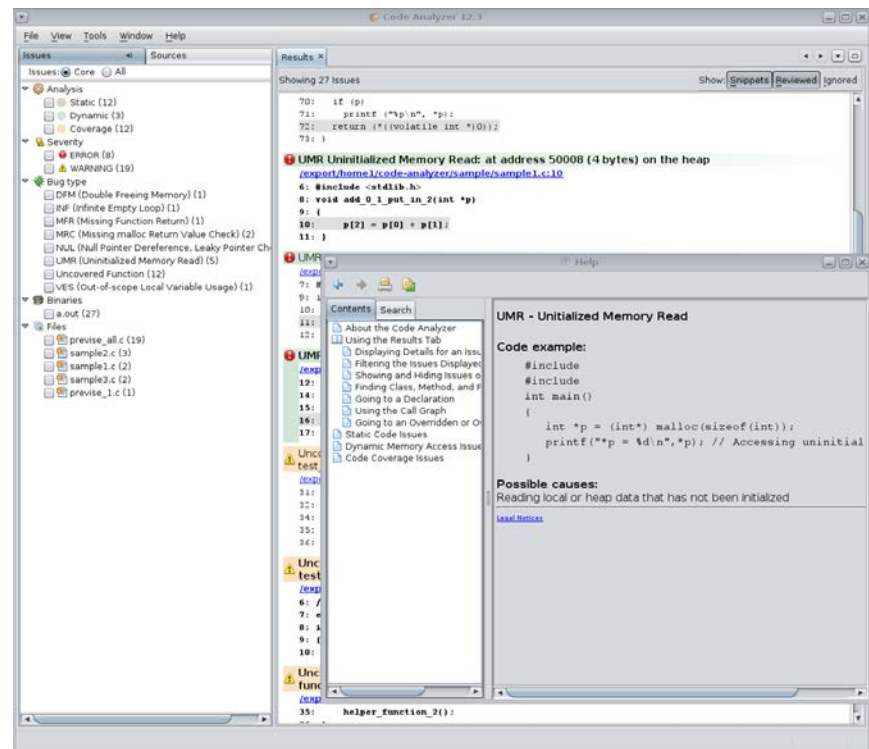


Figure 3: Overview of Code Analyzer GUI

The Code Analyzer combines the advantages of the bug-finding capabilities of static and dynamic code checking along with coverage capabilities to help developers produce better code with fewer errors in less time.

Contact Us

For more information about Oracle Solaris Studio, visit oracle.com/goto/solarisstudio or call +1.800.ORACLE1 to speak to an Oracle representative.



Oracle is committed to developing practices and products that help protect the environment

Copyright © 2005, 2010, 2011 Oracle and/or its affiliates. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark licensed through X/Open Company, Ltd. 0611

Hardware and Software, Engineered to Work Together