Oracle® Solaris Studio 12.3 Code Analyzer User's Guide

Part No: E23336
December 2011

ORACLE

120118@25097

Contents

Preface

The *Oracle Solaris Studio 12.3 Code Analyzer User's Guide* gives instructions on how to use the Code Analyzer tool, including collecting static, dynamic memory, and code coverage data with the compilers, Discover, and Uncover; and running the Code Analyzer GUI to analyze and display the data.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64" points out specific 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the Oracle Solaris Studio 12.3 Release Notes.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.
- Man pages for command-line tools describe a tool's command options.

Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

◆ ◆ ◆ CHAPTER 1

Introduction

The Oracle Solaris Studio Code Analyzer is an integrated set of tools designed to help developers of C and C++ applications for Oracle Solaris produce secure, robust, and quality software.

This chapter includes information about the following:

Data Analyzed by The Code Analyzer

The Code Analyzer analyzes three types of data:

- Static code errors detected during compilation
- Dynamic memory access errors and warnings detected by Discover, the memory error discovery tool
- Code coverage data measured by Uncover, the code coverage tool

In addition to giving you access to each individual type of analysis, the Code Analyzer integrates static code checking and dynamic memory access checking to add confidence levels to errors found in code. By using static code checking together with dynamic memory access analysis and code coverage analysis, you will be able to find many important errors in your applications that cannot be found by other error detection tools working by themselves.

The Code Analyzer also pinpoints the core issues in your code, those issues that, when fixed, are likely to eliminate the other issues. A core issue usually combines several other issues because, for example, the issues have a common allocation point, or occur at the same data address in the same function.

Static Code Checking

Static code checking detects common programming errors in your code during compilation. The `-xanalyze=code` option for the C and C++ compilers leverages the compilers' extensive and well proven control and data flow analysis frameworks to analyze your application for potential programming and security flaws.

For information on collecting static error data, see "Collecting Static Error Data" on page 13.

For a list of the static code errors the Code Analyzer analyzes, see "Static Code Issues" on page 19.

Dynamic Memory Access Checking

Memory-related errors in your code are often difficult to find. When you instrument your program with Discover before running it, Discover catches and reports memory access errors dynamically during program execution. For example, if your program allocates an array and does not initialize it, and then tries to read from a location in the array, the program is likely to behave erratically. If you instrument the program with Discover and then run it, Discover will catch the error.

For information on collecting dynamic memory access error data, see "Collecting Dynamic Memory Access Data" on page 14.

For a list of the dynamic memory access issues the Code Analyzer analyzes, see "Dynamic Memory Access Issues" on page 20.

Code Coverage Checking

Code coverage is an important part of software testing. It gives you information on which areas of your code are exercised in testing and which are not, enabling you to improve your test suites to test more of your code. The Code Analyzer uses data collected by Uncover to determine which functions in your program are uncovered and the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.

For information on collecting code coverage data, see "Collecting Code Coverage Data" on page 15.

Requirements for Using the Code Analyzer

The Code Analyzer works with static error data, dynamic memory access error data, and code coverage data collected from binaries compiled with the Oracle Solaris Studio 12.3 C or C++ compiler.

The Code Analyzer runs on a SPARC-based or x86–based system running the Solaris 10 10/08 operating system or a later Solaris 10 update, or Oracle Solaris 11.

The Code Analyzer GUI

After collecting data with the compiler, Discover, or Uncover, you can start the Code Analyzer GUI to display and analyze the issues.

For each issue, the Code Analyzer displays the issue description, the path name of the source file in which the issue was found, and a code snippet from that file with the relevant source line highlighted.

In the Code Analyzer, you can do the following:

- Display more details for an issue. For a static issue, the details include the Error Path. For a dynamic memory access issue, the details include a Call Stack, and if the data is available, also include an Allocation Stack and a Free Stack.
- Open the source file in which an issue was found.
- Jump from a function call in the Error Path or stack to the associated source code line.
- Find all of the usages of a function in your program.
- Jump to the declaration of a function.
- Jump to the declaration of an overridden or overriding funtion.
- Display the call graph for a function.
- Display more information about each issue type, including a code example, and possible causes.
- Filter the displayed issues by analysis type, issue type, and source file.
- Hide issues you have already reviewed, and close issues that you are not interested in.

For detailed information on using the GUI, see the online help in the GUI and the Oracle Solaris Studio 12.3 Code Analyzer Tutorial.

Quick Start

The following is an example of compiling a program to collect static code data, recompiling it with debug information, instrumenting it with Discover and running it to collect dynamic memory access data, instrumenting it with Uncover to collect code coverage data, and starting the Code Analyzer to display the collected data.

```
% cc -xanalyze=code *.c
% cc -g *.c
% cp a.out a.out.save
% discover -a a.out
% a.out
% cp a.out.save a.out
% uncover a.out
% a.out
% uncover -a a.out.uc
% code-analyzer a.out
```

◆ ◆ ◆ CHAPTER 2

Collecting Data And Starting the Code Analyzer

The data you collect for analysis by the Code Analyzer is stored in the *binary_name*`.analyze` directory in the directory that contains your source code files. The *binary_name*`.analyze` directory is created by the compiler, Discover, or Uncover, whichever one you run first to collect data on your program.

This chapter includes information about the following:

Collecting Static Error Data

To collect static error data on your C or C++ program, compile the program using the Oracle Solaris Studio 12.3 C or C++ compiler with the `-xanalyze=code` option. (The `-xanalyze=code` option is not available in the compilers in previous releases of Oracle Solaris Studio.) When you use this option, the compiler automatically extracts static errors and writes the data to the static subdirectory in the *binary_name*`.analyze` directory.

If you compile your program with the `-xanalyze=code` option and then link it in a separate step, you also need to include the `-xanalyze=code` option on the link step.

The compilers cannot detect all of the static errors in your code.

- Some errors depend on data that is available only at runtime. For example, given the following code, the compiler would not detect an ABW (beyond Array Bounds Write) error, because it could not detect that the value of `ix`, read from a file, lies outside the range [0,9]:

```
void f(int fd, int array[10])
{
  int ix;
```

```
    read(fd, &ix, sizeof(ix));
    array[ix] = 0;
}
```

- Some errors are ambiguous, that is, they might be real errors in your code, but they also might not be. The compiler does not report these errors.
- Some complex errors are not detected by the compilers in this release.

After collecting static error data, you can start the Code Analyzer GUI to analyze and display the data (see "Starting the Code Analyzer GUI" on page 16) or recompile the program so that you can collect dynamic memory access or code coverage data.

Collecting Dynamic Memory Access Data

Collecting dynamic memory access data on your C or C++ program is a two-step process: instrumenting the binary with Discover, and then running the instrumented binary.

To instrument your program with Discover to collect data for the Code Analyzer, you must have compiled the program with the Oracle Solaris Studio 12.3 C or C++ compiler. Compiling with the `-g` option generates debug information that allows the Code Analyzer to display source code and line number information for dynamic memory access errors and warnings.

Discover provides the most complete detection of memory errors at the source code level if you compile your program without optimization. If you compile with optimization, some memory errors will not be detected.

For information about specific types of binaries that Discover can or cannot instrument, see "Binaries That Redefine Standard Memory Allocation Functions Can Be Used" in *Oracle Solaris Studio 12.3: Discover and Uncover User's Guide* and "Binaries That Use Preloading or Auditing Cannot Be Used" in *Oracle Solaris Studio 12.3: Discover and Uncover User's Guide*.

Note – You can build your program once for use with both Discover and Uncover. But you cannot instrument a binary that is already instrumented, so if you are also planning to use Uncover to collect coverage data, save a copy of the binary for this purpose before instrumenting it with Discover. For example:

```
cp a.out a.out.save
```

To collect dynamic memory access data from the binary:

1. Instrument the binary with Discover using the `-a` option:

   `discover -a` *binary_name*

Note – You must use the version of Discover in Oracle Solaris Studio 12.3. The `-a` option is not available in earlier versions of Discover.

2. Run the instrumented binary. The dynamic memory access data is written to the `dynamic` subdirectory in the *binary_name*`.analyze` directory.

Note – For additional instrumentation options you can specify when instrumenting the binary with Discover, see "Instrumentation Options" in *Oracle Solaris Studio 12.3: Discover and Uncover User's Guide* or the `discover` man page. You can use the `-c`, `-F`, `-N`, or `-T` options with the `-a` option.

After collecting dynamic memory access data, you can start the Code Analyzer GUI to analyze and display the data, along with any static code data you might have previously collected (see "Starting the Code Analyzer GUI" on page 16. Or you can use an uninstrumented copy of the binary to collect code coverage data.

Collecting Code Coverage Data

Collecting code coverage data on your C or C++ program is a three-step process: instrumenting the binary with Uncover, running the instrumented binary, and then running Uncover again to generate a coverage report for use by the Code Analyzer.

You can run the instrumented binary multiple times after instrumenting it, and accumulate data over all of the runs before generating the coverage report

To instrument your program with Uncover to collect data for use by the Code Analyzer, you must have compiled the program with the Oracle Solaris Studio 12.3 C or C++ compiler. Compiling with the `-g` option generates debug information that allows the Code Analyzer to use source code level coverage information.

Note – If you saved a copy of the binary when you compiled your program for instrumenting with Discover, you can rename the copy to the original binary name and use it for instrumenting with Uncover. For example:

```
cp a.out.save a.out
```

To collect code coverage data from the binary:

1. Instrument the binary with Uncover:

   `uncover` *binary_name*

2. Run the instrumented binary one or more times. The code coverage data is written to a *binary_name*`.uc` directory.
3. Generate the code coverage report from the accumulated data using Uncover with the `-a` option:

   ```
   uncover -a  binary_name.uc
   ```

   The coverage report is written to the coverage subdirectory in the *binary_name*`.analyze` directory.

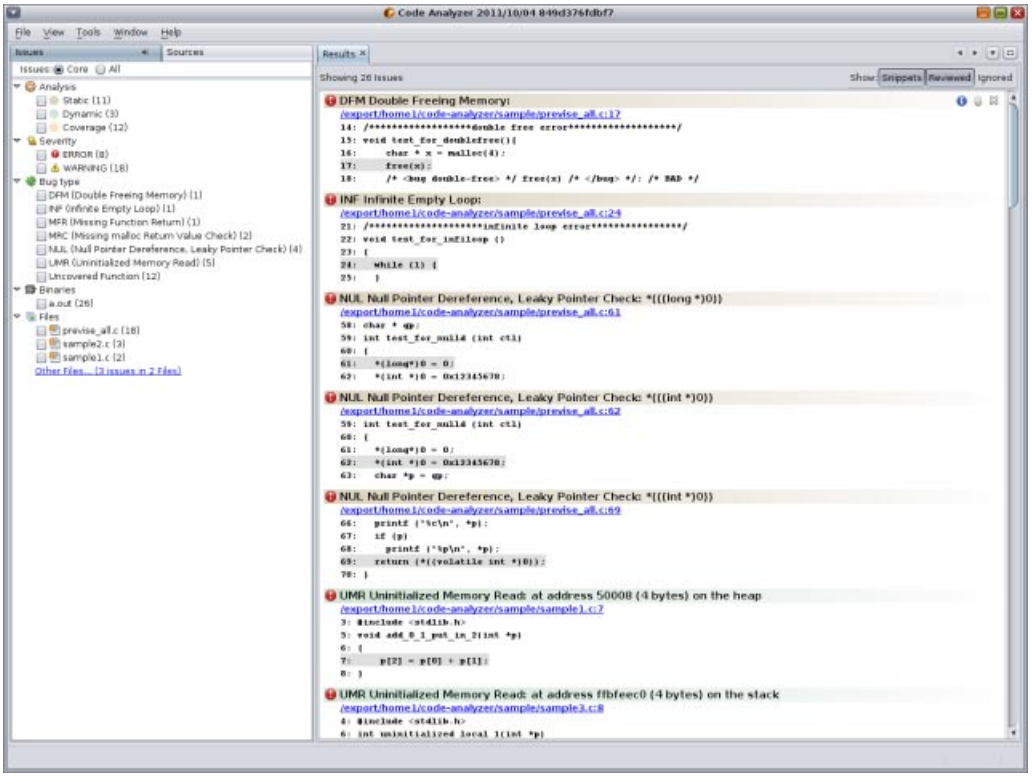
Note – You must use the version of Uncover in Oracle Solaris Studio 12.3. The `-a` option is not available in earlier versions of Uncover.

Starting the Code Analyzer GUI

You can use the Code Analyzer GUI to analyze one, two, or all three types of data. To start the GUI, type the `code-analyzer` command and the path to the binary for which you want to analyze error data you have collected:

```
code-analyzer binary_name
```

The Code Analyzer GUI opens and displays the data in the *binary_name*`.analyze` directory.



When the Code Analyzer GUI is running, you can switch to displaying the data you have collected for a different binary by choosing Open > File and navigating to the binary.

The online help in the GUI describes how to use all of features to filter the displayed results, show or hide issues, and show more information about specific issues. The Oracle Solaris Studio 12.3 Code Analyzer Tutorial guides you through a complete scenario of data collection and analysis using a sample program.

◆ ◆ ◆ APPENDIX A

Errors Analyzed by the Code Analyzer

The compilers, Discover, and Uncover find static code issues, dynamic memory access issues, and coverage issues in your code. The specific error types that are found by these tools and analyzed by the Code Analyzer are listed in the following sections.

Static Code Issues

Static code checking finds the following types of errors:

- ABR: beyond Array Bounds Read
- ABW: beyond Array Bounds Write
- DFM: Double Freeing Memory
- ECV: Explicit type Cast Violation
- FMR: Freed Memory Read
- FMW: Freed Memory Write
- INF: INFinite empty loop
- Memory leak
- MFR: Missing Function Return
- MRC: Missing malloc Return value Check
- NFR: uNinitialized Function Return
- NUL: NULl pointer dereference, leaky pointer check
- RFM: Return Freed Memory
- UMR: Uninitialized Memory Read, Uninitialized Memoey Read bit operation
- URV: Unused Return Value
- VES: out-of-scope local Variable usage

Dynamic Memory Access Issues

Dynamic memory access checking finds the following types of errors:

- ABR: beyond Array Bounds Read
- ABW: beyond Array Bounds Write
- BFM: Bad Free Memory
- BRP: Bad Realloc address Parameter
- CGB: Corrupted Guard Block
- DFM: Double Freeing Memory
- FMR: Freed Memory Read
- FMW: Freed Memory Write
- IMR: Invalid Memory Read
- IMW: Invalid Memory Write
- Memory leak
- OLP: OverLaPping source and destination
- PIR: Partially Initialized Read
- SBR: beyond Stack Bounds Read
- SBW: beyond Stack Bounds Write
- UAR: UnAllocated memory Read
- UAW: UnAllocated memory Write
- UMR: Unitialized Memory Read

Dynamic memory access checking finds the following types of warnings:

- AZS: Allocating Zero Size
- Memory leak
- SMR: Speculative unitialized Memory Read

Code Coverage Issues

Code coverage checking determines which functions are uncovered. In the results, code coverage issues found are labeled as Uncovered Function, with a potential coverage percentage, which indicates the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.

Index

X