Oracle® Developer Studio 12.5: Discover and Uncover User's Guide

Part No: E60755
June 2016

ORACLE

Oracle Developer Studio 12.5: Discover and Uncover User's Guide

Part No: E60755

Access to Oracle Support

Oracle customers that have purchased support 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.

Référence: E60755

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Using This Documentation

- **Overview** – Describes how to use the Memory Error Discovery Tool (`discover`) to find memory-related errors in binaries, and the Code Coverage Tool (`uncover`) to measure code coverage of applications.
- **Audience** – Application developers, system developers, architects, support engineers
- **Required knowledge** – Programming experience, software development testing, experience in building and compiling software products

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E60778-01`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ CHAPTER 1

Introduction

Oracle Developer Studio 12.5 Discover and Uncover User's Guide describes how to use the following tools:

Memory Error Discovery Tool (`discover`)

The Memory Error Discovery Tool (`discover`) software is an advanced development tool for detecting memory access errors. The `discover` utility works on binaries compiled with Sun Studio 12 Update 1, Oracle Solaris Studio 12.2-12.4 or Oracle Developer Studio 12.5 compilers. It works on a SPARC–based or x86–based system running at least one of the following operating systems: Solaris 10 10/11, Oracle Solaris 11.3, Oracle Enterprise Linux 6.x, or Oracle Enterprise Linux 7.x.

Memory-related errors in programs are notoriously difficult to find. The `discover` utility enables you to find such errors easily by pointing out the exact place where the problem exists in the source code. For example, if your program allocates an array but does not initialize it, and then tries to read from one of the array locations, the program will probably behave erratically. The `discover` utility can catch this problem when you run the program in the normal way.

Other errors detected by `discover` include:

- Reading from and writing to unallocated memory
- Accessing memory beyond allocated array bounds
- Incorrect use of freed memory
- Freeing the wrong memory blocks
- Freeing the same memory block multiple times
- Memory leaks
- Overlapping memory copy

- Stale pointer accesses
- Incorrect parameters to system library functions

Because `discover` catches and reports memory access errors dynamically during program execution, if a portion of user code is not executed at runtime, errors in that portion are not reported.

The `discover` utility is simple to use. Any binary (even a fully optimized binary) can be instrumented with a single command, then run in the normal way. For information on how best to instrument your binary, see "Supported Binaries" on page 13. During the run, `discover` produces a report of the memory anomalies, which you can view as a text file or as HTML in a web browser.

Code Coverage Tool (`uncover`)

The `uncover` utility is a simple and easy to use command-line tool for measuring code coverage of applications. Code coverage is an important part of software testing. It provides information about which areas of your code are exercised in testing, enabling you to improve your test suites to test more of your code. The coverage information that `uncover` reports can be at a function, statement, basic block, or instruction level.

The `uncover` utility provides a unique feature called *uncoverage*, which enables you to quickly find major functional areas that are not being tested. Other advantages of `uncover` code coverage are:

- The slowdown relative to uninstrumented code is fairly small.
- Because `uncover` operates on binaries, it can work with any optimized binary.
- Measurements can be done simply by instrumenting the shipping binary. You do not have to build the application differently for coverage testing.
- The `uncover` utility provides a simple procedure for instrumenting the binary, running tests, and displaying the results.
- The `uncover` utility is multithread safe and multiprocess safe.

♦ ♦ ♦ CHAPTER 2

Memory Error Discovery Tool (`discover`)

The Memory Error Discovery Tool (`discover`) software is an advanced development tool for detecting memory access errors.

This chapter includes information about the following:

Requirements for Using `discover`

This section describes requirements for using `discover` and achieving the best results and contains the following topics:

Supported Binaries

The `discover` utility works on binaries compiled with Sun Studio 12 Update 1, Oracle Solaris Studio 12.2-12.4 or Oracle Developer Studio 12.5 compilers. It works on a SPARC–based or

x86–based system running at least one of the following operating systems: Solaris 10 10/08, Oracle Solaris 11, Oracle Enterprise Linux 5.x, or Oracle Enterprise Linux 6.x.

The `discover` utility issues an error and does not instrument a binary if it does not meet these requirements. However, you can instrument a binary that does not meet these requirements and use the `-l` option to detect a limited number of errors. See "Instrumentation Options" on page 20.

A compiled binary includes information called annotations to help `discover` instrument it correctly. The addition of this small amount of information does not affect the performance of the binary or its runtime memory usage.

Use the `-g` option to generate debug information when compiling the binary so `discover` can display source code and line number information while reporting errors and warnings, and produce more accurate results. If your binary is not compiled with the `-g` option, `discover` displays only the program counters of the corresponding machine level instructions. Also, compiling with the `-g` option helps `discover` produce more accurate reports. While `discover` can work with many optimized binaries, the use of `-g` is still recommended. For more information, see "Interpreting `discover` Error Messages" on page 56.

For best results, binaries should be compiled with no optimization options and with the `-g` option. Optimized code can vary from the source code due to optimizations, such as use of same memory locations for different variables and generation of speculative code. Using advanced optimization options while compiling can cause `discover` to report incorrect errors or fail to report errors.

Note - `discover` supports binaries that redefine the standard memory allocation functions: `malloc()`, `calloc()`, `memalign()`, `valloc()`, and `free()`.

For more information, see "Limitations When Using `discover`" on page 59.

Binaries That Use Preloading or Auditing Are Incompatible

Because `discover` uses some special features of the runtime linker, you cannot use it with binaries that use preloading or auditing.

If a program requires the setting of the `LD_PRELOAD` environment variable, it probably will not work correctly with `discover` because `discover` needs to interpose on certain system functions, and it cannot do so if the function has been preloaded.

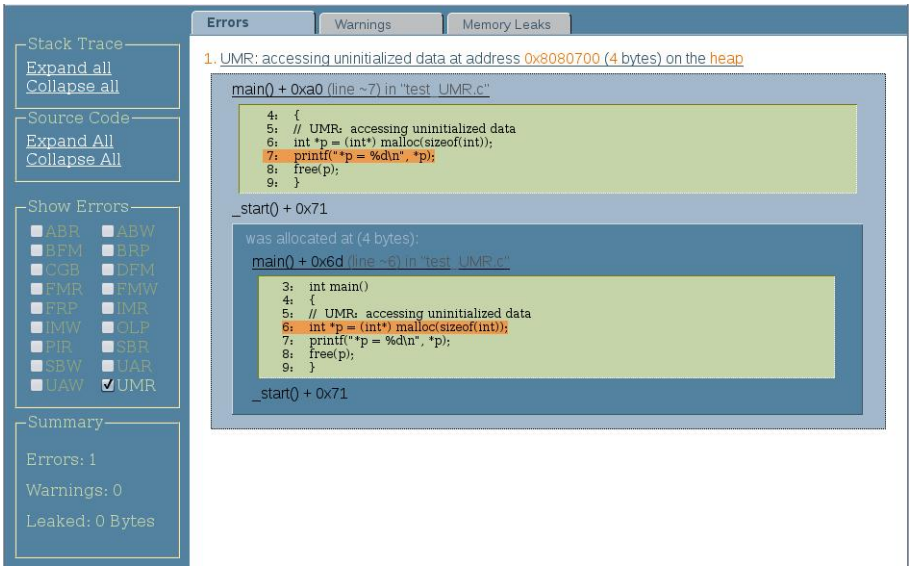
Similarly, if a program uses runtime auditing, either because the binary was linked with the `-p` option or the `-P` option or it requires the `LD_AUDIT` environment variable to be set, this auditing will conflict with `discover`'s use of auditing. If the binary was linked with auditing, `discover` fails at instrumentation time. If you set the `LD_AUDIT` environment variable at runtime, the results are undefined.

Simple Program Example

The following example illustrated preparing a program, instrumenting it with `discover`, and then running it and producing a report on the detected memory access errors. This example uses a simple program that accesses uninitialized data.

```
% cat test_UMR.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
//  UMR:  accessing uninitialized data
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n", *p);
free(p);
}

% cc -g  test_UMR.c
% a.out
*p = 131464
% discover a.out
% a.out
```

The `discover` output indicates where the uninitialized memory was used and where it was allocated, along with a summary of results, as shown in the following figure.



Instrumenting a Binary

Instrumenting a targeted binary adds code in strategic places so that `discover` can keep track of memory operations while the binary is running.

Note - For 32-bit binary on SPARC V8 architecture, `discover` inserts V8plus code while instrumenting. As a result, the output binary is always v8plus regardless of binary input.

You instrument a binary using the `discover` command. For example, the following command instruments the binary `a.out` and overwrites the input `a.out` with the instrumented `a.out`:

```
discover a.out
```

When you run the instrumented binary, `discover` monitors the program's use of memory. During the run, `discover` writes a report detailing any memory access errors to an HTML file that you can view in your web browser. The default file name is `a.out.html`. To request that the report be written to an ASCII file or to `stderr`, use the `-w` option when you instrument the binary.

When `discover` instruments a binary, if it finds any code that it cannot instrument because it is not annotated, it displays a warning like the following:

```
discover: (warning): a.out: 80% of code instrumented (16 out of 20 functions)
```

Non-annotated code could come from assembly language code linked into the binary, or from modules compiled with compilers or on operating systems older than those listed in “Supported Binaries” on page 13.

Caching Shared Libraries

When `discover` instruments a binary, it adds code to the binary that works with the runtime linker to instrument dependent shared libraries when they are loaded at runtime. The instrumented libraries are stored in a cache where they can be reused if the original has not changed since it was last instrumented. By default, the cache directory is `$HOME/SUNW_Bit_Cache`. You can change the directory with the `-D` option.

Instrumenting Shared Libraries

The `discover` utility produces the most accurate results if the entire program, including all shared libraries, is instrumented. By default, `discover` checks and reports memory errors only in executables. To specify that you want `discover` to skip checking for errors in executables, use the `-n` option.

You can use the `-c` <lib> option to specify that you want `discover` to check for errors in the dependent shared libraries and libraries dynamically opened by `dlopen()`. You can also use the `-c` option to avoid checking for errors in a specific library. Although `discover` does not report any errors in that library, because it needs to track the memory state of the entire address space to correctly detect memory errors, it records allocations and memory initializations in the entire program including all shared libraries.

The `discover` utility runtime uses the linker audit interface, also called the `rtld-audit` or `LD_AUDIT` to automatically load instrumented shared libraries from `discover`'s cache directory. On Oracle Solaris, the audit interface is used by default. On Linux, you need to set `LD_AUDIT` on the command line while running the instrumented binary.

For 32-bit applications on Oracle Linux:

```
% LD_AUDIT=install-dir/lib/compilers/bitdl.so a.out
```

For 64-bit applications on Oracle Linux:

```
% LD_AUDIT=install-dir/lib/compilers/amd64/bitdl.so a.out
```

This mechanism might not work in all environments running Oracle Enterprise Linux 5.x. If no library instrumentation is needed and `LD_AUDIT` is not set, there `discover` has no issues on Oracle Enterprise Linux 5.x.

You should instrument all shared libraries used by the program as described in "Instrumenting a Binary" on page 16. By default, if the runtime linker encounters an uninstrumented library, a fatal error occurs. You can, however, tell `discover` to ignore one or more libraries.

Ignoring Libraries

You might not be able to instrument some libraries . You can tell `discover` to ignore these libraries with the `-T` or `-N` option (see "Instrumentation Options" on page 20) or with specifications in `bit.rc` files (see "`bit.rc` Initialization Files" on page 22). Some accuracy might be lost.

By default, `discover` uses specifications in the system `bit.rc` file to set certain system and compiler-supplied libraries to be ignored because they may not be annotated. The effect on accuracy is minimal because `discover` knows the memory characteristics of the most commonly used libraries.

Checking Parts of a Library or an Executable

You can specify an executable or library using the `-c` option. You can further qualify a target executable or target library by restricting the memory access checking to certain object files.

For example, if the target library is `libx.so` and the target executable is `a.out`, you would use the following command:

```
$ discover -c libx.so -o a.out.disc a.out
```

You can also limit the checking of any target by adding colon-separated files or directories. Files can be ELF files or directories. If you specify an ELF file, all functions defined in the file are checked. If you specify a directory, all files in the directory are recursively used.

```
$ discover -o a.out.disc a.out:t1.0:dir
```

```
$ discover -c libx.so:l1.o:12.o -o a.out.disc a.out
```

Command-Line Options

You can use the following options with the `discover` command to instrument a binary.

Output Options

`-a` — Write the error data to *binary-name*`.analyze/dynamic` directory for use by Code Analyzer.

`-b` *browser* — Start web browser *browser* automatically while running the instrumented program (`off` by default).

`-e` *n* — Show only *n* memory errors in the report (default is show all errors).

`-E` *n* — Show only *n* memory leaks in the report (default is 100).

`-f` — Show offsets in the report (default is to hide them).

`-H` *html-file* — Write `discover`'s report on the binary in HTML format to *html-file*. This file is created when you run the instrumented binary. If *html-file* is a relative pathname, it is placed relative to the working directory where you run the instrumented binary. To make the file name unique for each time you run the binary, add the string `%p` to the filename to instruct the `discover` runtime to include the process ID. For example, the option `-H report.%p.html` generates a report file with the file name `report.`*process-ID*`.html`. If you include `%p` in the file name more than once, only the first instance is replaced with the process ID.

If you do not specify this option or the `-w` option, the report is written in HTML format to *output-file*`.html`, where *output-file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

You can specify both this option and the `-w` option to write the report in both text and HTML formats.

`-m` — Show mangled names in the report (default is to show demangled names).

`-o` *file* — Write the instrumented binary to *file*. By default, the instrumented binary overwrites the input binary.

`-S` *n* — Show only *n* stack frames in the report (default is 8).

`-w` *text-file* — Write `discover`'s report on the binary to *text-file*. The file is created when you run the instrumented binary. If *text-file* is a relative pathname, the file is placed relative to the working directory where you run the instrumented binary. To make the file name unique for each time you run the binary, add the string `%p` to the file name to ask the `discover` runtime

to include the process ID. For example, the option `-w report.%p.txt` generates a report file with the file name `report.`*process-ID*`.txt`. If you include `%p` in the file name more than once, only the first instance is replaced with the process ID. Specifying `-w -` will output in `stderr`.

If you do not specify this option or the `-H` option, the report is written in HTML format to *output-file*`.html`, where *output-file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

You can specify both this option and the `-H` option to write the report in both text and HTML formats.

Note - Using the full path names is recommended while using the `-w` and `-H` options. If relative paths are used, the reports are generated in the directory relative to the run directory of the process. So if the application changes directories and starts new processes, it is possible that the reports might be misplaced. When applications fork new processes, `discover` at runtime makes a copy of parent error report for the child process and the child process continues to write to the copy. If the run directory of the child process is different, and a relative path was used for the report file, it is possible that the child process will not find the parent process. Using the full path name prevents these issues.

Instrumentation Options

`-A [on | off]`

Turn on or off allocation/free stack traces (default is `on` with stack depth 8). This flag can only be specified when instrumenting for hardware-assisted checking, using the `-i adi` option. For better runtime performance, allocation/free stack trace collection can be turned off with this option. This option can only be used if you have Oracle Developer Studio 12.5 , 3/15 Platform Specific Enhancement (PSE) installed.

`-c [ - |` *library [: scope...]* `|` *file*`]`

Check for errors in all libraries, or in the specified *library*, or in the libraries listed separated by new lines in the specified *file*. The default is not to check for errors in libraries. You can limit the scope of checking the library by adding colon-separated files or directories. Some critical errors like bad memory write errors may still be reported even if the `--c` flag is used since these errors could potentially corrupt the memory that belongs to other binaries in the application. For more information, see "Checking Parts of a Library or an Executable" on page 18.

`-F [parent | child | both]`

Specify what you want to happen if a binary you have instrumented with `discover` forks while you are running it. By default, `discover` continues

to collect memory access error data from both parent and child processes. If you want `discover` to follow only the parent process, specify `-F parent`. If you want Discover to follow only the child process, specify `-F child`.

`-i [datarace | memcheck | adi]`

Determines instrumentation type of `discover` (default is `memcheck`).

If `datarace` is specified, instrument for data race detection using Thread Analyzer. When you use this option, only data race detection is done at runtime; no other memory checking is done. The instrumented binary must be run with the `collect` command to generate an experiment that you can view in Performance Analyzer. For more information, see *Oracle Developer Studio 12.5: Thread Analyzer User's Guide*.If `memcheck` is specified, instrument for memory error checking. If `adi` is specified, instrument for hardware-assisted checking using the SPARC M7 processor ADI feature. This feature is only available with Oracle Solaris 11.3 running on SPARC M7 processor.

`-K`

Do not read the `bit.rc` initialization files (see "`bit.rc` Initialization Files" on page 22).

`-l`

Run `discover` in lite mode. Use the `-l` optionif you are only interested in finding memory leaks in your program. This mode can also identify some memory access errors without slowing down the program much. Examples of such errors are `double free of a memory area` and `out of bounds access of an allocated area` passed as an argument to a `memcpy()` function call. It is recommended that you run your program through lite mode in `discover` before running it in full mode.

`-n`

Do not check for errors in executables. Some critical errors like memory write errors may still be reported even if the `--n` flag is used, because these errors could potentially corrupt the memory that belongs to other binaries in the application.

`-N` *library*

Do not instrument any dependent shared library matching the prefix *library*. If the initial characters of a library name match *library*, the library is ignored. If *library* begins with a slash (/), matching is done on the full absolute pathname of the library. Otherwise, matching is done on the basename of the library.

`-P [on | off]`

Turn on or off precise ADI mode. Default is `on`. This flag can only be specified when instrumenting for hardware-assisted checking, using the `-i adi` option. For better runtime performance, you can turn off precise ADI mode with this option.

`-T`	Instrument the named binary only. Do not instrument any dependent shared libraries at runtime.

Caching Options

`-D` *cache-directory*	Use *cache-directory* as the root directory for storing cached instrumented binaries. By default, the cache directory is `$HOME/SUNW_Bit_Cache`.
`-k`	Force reinstrumentation of any libraries found in the cache.

Other Options

`-h` or `-?`	Help. Print a short usage message and exit.
`-v`	Verbose. Print a log of what `discover` is doing. Specify the option twice for more information.
`-V`	Print `discover` version information and exit.

`bit.rc` Initialization Files

The `discover` utility initializes its state by reading a series of `bit.rc` files at startup. A system file, *Oracle-Developer-Studio-installation-directory*`/lib/compilers/bit.rc`, provides default values for certain variables. The `discover` utility reads this file first, followed by `$HOME/.bit.rc` if it exists, and *current-directory*`/.bit.rc` if it exists.

The `bit.rc` files contain commands to set, append, or remove certain variable values. When `discover` reads a `set` command, it discards the previous value, if any, of the variable. When it reads an `append` command, it appends the argument (after a colon separator) to the existing value of the variable. When it reads a `remove` command, it removes the argument and its colon separator from the existing value of the variable.

The variables set in the `bit.rc` files include the list of libraries to ignore when instrumenting, and lists of functions or function prefixes to ignore when computing the percentage of non-annotated code in a binary.

For more information, refer to the comments in the header of the system `bit.rc` file.

Running an Instrumented Binary

After you have instrumented your binary with `discover`, you run the binary the same way you normally would. Typically, if a particular combination of input causes your program to behave unexpectedly, you would instrument it with `discover` and run it with the same input to investigate potential memory problems. While the instrumented program is running, `discover` writes information about any memory problems it finds to the specified output files in the selected formats (text, HTML, or both). For information about interpreting the reports, see "Analyzing `discover` Reports" on page 35.

Because of the overhead of instrumentation, your program is likely to run significantly slower after you instrument it. Depending on the frequency of memory access, it might run as much as 50 times slower.

Hardware-Assisted Checking Using Silicon Secured Memory (SSM)

The SPARC M7 processor from Oracle offers Software in Silicon, which enables software to run faster and more reliably. One Software in Silicon feature is Silicon Secured Memory (SSM), previously called Application Data Integrity (ADI), whose circuitry detects common memory access errors that can cause run-time data corruption.

These errors can be caused by errant code or a malicious attack on a server's memory. For example, buffer overflows are known to be a major source of security exploits. Further, in-memory databases increase an application's exposure to such errors due to having critical data in-memory.

Silicon Secured Memory stops memory corruptions in optimized production code by adding version numbers to the application's memory pointers and the memory they point to. If the pointer version number does not match the content version number, the memory access is aborted. Silicon Secured Memory works with applications written in systems-level programming languages such as C or C++, which are more vulnerable to memory corruption caused by software errors.

Oracle Developer Studio 12.5 includes the `libdiscoverADI.so` library (also referred to as the `discover` ADI library), which provides updated `malloc()` library routines that ensure that adjacent data structures are given different version numbers. These version numbers enable the processor's SSM technology to detect memory access errors, like buffer overflows. Memory content version numbers are changed when memory structures are freed to prevent stale pointer

accesses. For more information about the errors caught by `discover` and `libdiscoverADI.so`, see “Errors Caught by the `libdiscoverADI` Library” on page 24.

In addition to using SSM in production to detect potential memory corruption issues, , you can use it during application development to ensure that such errors are caught during application testing and certification. Memory corruption bugs are extremely hard to find because applications encounter corrupted data long after the corruption happens. The `discover` tool and the `libdiscoverADI.so` library, part of the Oracle Developer Studio developer tool suite, provide you with additional application information that makes locating and fixing the errant code easier.

Using the `libdiscoverADI` Library to Find Memory Access Errors

The `discover` ADI library `libdiscoverADI.so` reports programming errors that result in invalid memory accesses. You can use in it two ways:

- By preloading the `discover` ADI library into your application with the `LD_PRELOAD_64`. environment variable. This method runs all 64-bit binaries in the application in ADI mode, For example, if you normally run an application named `server`, the command would be as follows:

  ```
  $ LD_PRELOAD_64=install-dir/lib/compilers/sparcv9/libdiscoverADI.so server
  ```

- By using ADI mode with the `discover` command with the `-i adi` option on a specific binary.

  ```
  % discover -i adi a.out
  % a.out
  ```

The errors are reported in an `a.out.html` file by default. For more information about `discover` reports, see “Analyzing `discover` Reports” on page 35 and “Output Options” on page 19.

See “Requirements and Limitations of Using `libdiscoverADI`” on page 30.

Errors Caught by the `libdiscoverADI` Library

The `libdiscoverADI.so` library catches the following errors:

- Array out of Bounds Access (ABR/ABW)

- Freed Memory Access (FMR/FMW)
- Stale Pointer Access (A special type of FMR/FMW)
- Unallocated Read/Write (UAR/UAW)
- Double Free Memory (DFM)

For more information about each of these types of errors, see "Memory Access Errors and Warnings" on page 50.

For a full example, see "Example of Using `discover` ADI Mode" on page 31.

Instrumentation Options for `discover` ADI Mode

The following options determine the precision and amount of information generated in the `discover` report when instrumenting with ADI.

`-A [on | off]`

When this flag is set to `on`, the `discover` ADI library reports the location of the error and the error stack trace. This information is sufficient to catch the error, but it is not always sufficient to fix the error. See "`SUNW_DISCOVER_OPTIONS` Environment Variable" on page 50 on how to change runtime behavior for this option.

This flag also generates information about where the offending memory area was allocated and freed. For example, the output might say that an error was an `Array out of Bounds Access` and where that array was allocated. If set to `off`, allocations and stack trace are not reported. The default is `on`.

Note - Even if `-A` is set to `on`, it is possible that ABR/ABW might sometimes be reported as FMR/FMW or UAR/UAW, due to one of the following reasons:

- If the buffer overflow access happens at a large offset after the end of the buffer or before the beginning of the buffer.
- If `libdiscoverADI.so` hits a resource limit. In this case, `discover` might be able to keep the allocation stack trace that is needed to determine if the error is a buffer overflow.

`-P [on | off]`

When this flag is set to `off`, ADI is run in non-precise mode. In non-precise mode, memory write errors are caught a few instructions (source lines) after the exact instruction is executed. To enable Precise mode, set this flag to `on`, which is the default.

For better runtime performance, you can specify `-A off`, `-P off`, or both options can be set to `off`.

Custom Memory Allocators and the `discover` ADI Library

Enterprise applications can often manage their own memory and do not use the system (`libc`) `malloc`(3C) library. For these cases, the normal usage of `libdiscoverADI.so`, which interposes on `malloc()` will not be able to catch memory corruptions. Oracle Developer Studio provides APIs so the enterprise applications can tell `libdiscoverADI.so` when a memory area is allocated or freed. `libdiscoverADI.so` manages the SSM versioning, signal handling, and error reporting. These APIs and `libdiscoverADI.so` only works for memory allocated by `mmap`(2) or `shmget`(2).

The following APIs are provided:

`void *adi_mark_malloc (void *, size_t);`	The allocator passes a pointer to the memory about to be passed and the size to the client requesting memory. If the memory does not have ADI enabled, the library uses a `memcntl`(2) call. A versioned pointer is returned, which the allocator must pass to the client.
void adi_mark_free (void *, size_t);	The allocator passes a pointer and the size to the memory area to be freed.
void *adi_unversion (void *);	Allocators can use pointer arithmetic to access a client memory's meta data. Client memory is often versioned, but allocator meta data is not. In such circumstances, you need to use this API. The allocator passes any pointer, and gets back an equivalent unversioned pointer.
void *adi_version (void *);	Sometimes allocators lose track of versioned pointers. This API takes in any pointer, and returns an equivalent pointer with the correct version on it. Any dereference with the return value will not cause an ADI SEGV.
void *adi_clear_version (void *, size_t);	If a memory area is being reused for a different purpose, the ADI version needs to be cleared. For example, if a memory area was versioned for use by an allocator client, and is now being used by the allocator for meta-data. This function takes in a versioned or unversioned pointer and size, sets the version for that area to 0, and returns an unversioned pointer.

Your application might manage its own memory allocation and free lists, for example by allocating large chunks of memory and subdividing it in your program. See Using Application Data Integrity and Oracle Solaris Studio to Find and Fix Memory Access Errors (`https://community.oracle.com/docs/DOC-912448`) for information about how you can use the ADI versioning APIs to catch errors with your managed memory.

Using Custom Memory Allocators

To use these APIs, include the following header file in your sources:

install-dir`/lib/compilers/include/cc/discoverADI_API.h`

Then, do one of the following:

- Link your sources with *install-dir*`/lib/compilers/sparcV9/libadiplugin.so`
- Set the environment variable `LD_PRELOAD_64` to *install-dir*`/lib/compilers/sparcV9/libadiplugin.so`
- Run the following command: `discover -i adi` *executable*

The following is an example of using custom memory allocators.

EXAMPLE 1 Using Custom Memory Allocators

The following is an example header file:

```
% cat allocator.h
#define GRANULARITY 32

void *mymalloc(size_t len);
void myfree(void *ptr);
```

The following is example source code, using custom memory allocators and the `libdiscoverADI` library:

```
% cat allocator.c
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <errno.h>
#include <dlfcn.h>
#include <unistd.h>
#include <assert.h>

#include "allocator.h"

#include "discoverADI_API.h"

#pragma init (setup_allocator)
#pragma fini (takedown_allocator)

#define MAX_ALLOCATIONS 1024
```

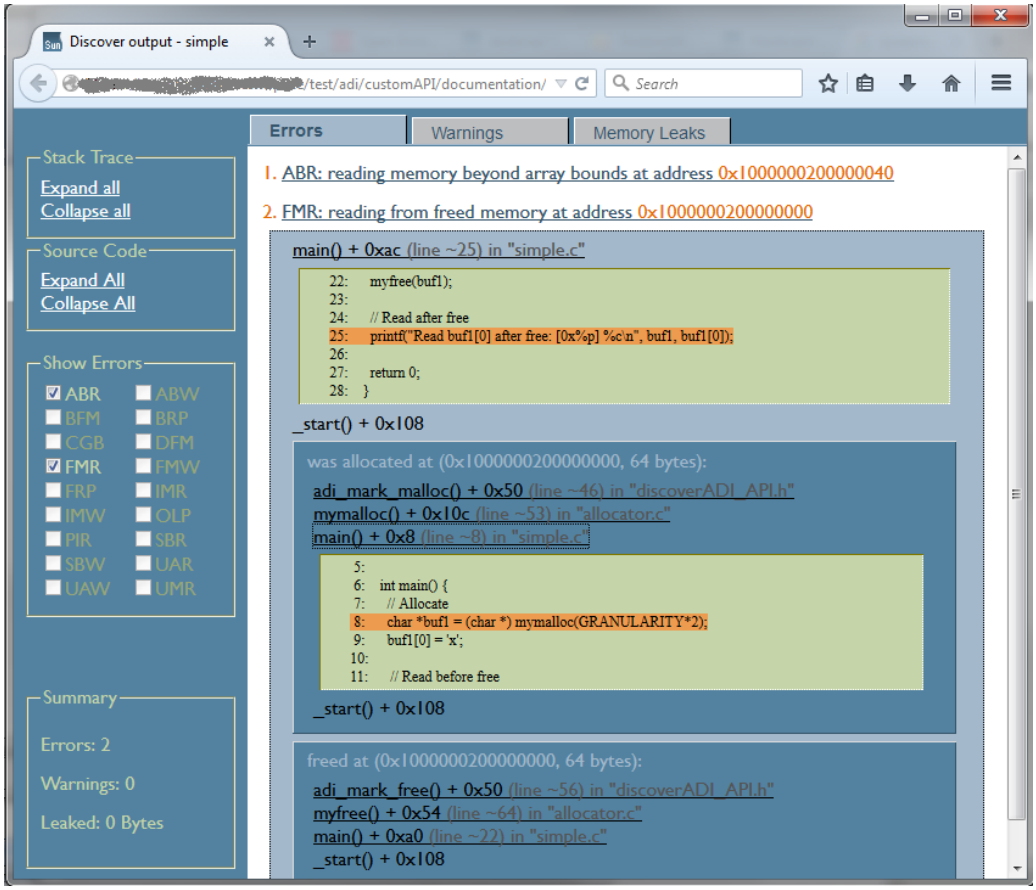
```
#define START_ADDRESS 0x200000000

uint64_t next_available_address = 0;
size_t allocation_table[MAX_ALLOCATIONS/GRANULARITY];
static void setup_allocator() {
  // mmap with a specific address
  void *addr = mmap((void *) START_ADDRESS, MAX_ALLOCATIONS,
                    PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
  if (addr == MAP_FAILED) {
    fprintf(stderr, "mmap failed {%d}\n", errno);
    exit(1);
  }

  next_available_address = (uint64_t) addr;
}

static void takedown_allocator() {
  if (munmap((void *) START_ADDRESS, MAX_ALLOCATIONS) != 0) {
    fprintf(stderr, "munmap failed {%d}\n", errno);
    exit(1);
  }
}
// Simple malloc
void *mymalloc(size_t size) {

  void *vaddr = (void *) next_available_address;
  next_available_address += size;

  assert(next_available_address < (START_ADDRESS+MAX_ALLOCATIONS));

  int index = ((uint64_t)vaddr-START_ADDRESS)/GRANULARITY;
  allocation_table[index] = size;

  // Tell libdiscoverADI.so about the allocation, get a versioned
  // pointer
  vaddr = adi_mark_malloc(vaddr, size);

  // Return the versioned pointer
  return vaddr;
}

// Simple free
void myfree(void *ptr) {
  int index = ((uint64_t)ptr-START_ADDRESS)/GRANULARITY;
  size_t size = allocation_table[index];

  // Tell libdiscoverADI.so about the free().
  adi_mark_free(ptr, size);
```

```
}

% cat simple.c
#include <stdio.h>
#include <stdlib.h>

#include "allocator.h"

int main() {
  // Allocate
  char *buf1 = (char *) mymalloc(GRANULARITY*2);
  buf1[0] = 'x';

  // Read before free
  printf("Read buf1[0] before free: [0x%p] %c\n", buf1, buf1[0]);

  // Allocate
  char *buf2 = (char *) mymalloc(GRANULARITY*2);

  // Buf1fer overflow buf1
  printf("Read buf1[%d] past end: [0x%p] %c\n", GRANULARITY*2,
  buf1+GRANULARITY*2, buf1[GRANULARITY*2]);

  // Free
  myfree(buf1);

  // Read after free
  printf("Read buf1[0] after free: [0x%p] %c\n", buf1, buf1[0]);

  return 0;
}
```

To compile and run this example:

```
% cc –m64 –g –I install-dir/lib/compilers/include/cc allocator.c simple.c install-dir/lib/
compilers/sparcv9/libadiplugin.so –o simple
% ./simple
```

The following is the resulting generated HTML report:



Requirements and Limitations of Using `libdiscoverADI`

You can use ADI mode of `discover` only with 64-bit applications on a SPARC M7 chip running at least Oracle Solaris 11.2.8 or Oracle Solaris 11.3.

Similar to instrumenting for memory checking, preloaded libraries might conflict if functions of `libdiscoverADI.so` interpose on the same allocation functions. See "Binaries That Use Preloading or Auditing Are Incompatible" on page 14 for more information.

Other limitations for checking your code with `libdiscoverADI` include the following:

- Only heap-checking is available. There is no stack checking, no static array-out-of-bounds checking, and no leak detection.
- Does not work with applications which use the unused bits in 64-bit addresses for storing meta data. Some 64-bit applications might use the currently unused high bits in 64-bit addresses for storing meta-data, for instance, locks. Such applications will not work with `discover` in ADI mode because the feature works by using the 4 highest bits in the 64-bit address to store version information.
- Might not work for applications that do pointer arithmetic with assumptions about heap addresses, for example, the distance between two successive allocations.
- Unlike in `memcheck` mode (instrumenting with `-i memcheck`), ADI mode does not catch errors if the application redefines standard memory allocation functions in the executable. If the application redefines the standard memory allocation functions in a library, then ADI mode works.
- Resolution for buffer overflow is 64 bytes. For allocations that are 64-byte-aligned, `libdiscoverADI.so` will catch any overflow by 1 byte or more. For allocations that are not aligned at 64 bytes, it might miss the buffer overflow by a few bytes. In general, overflow by 1 to 63 bytes might not be caught depending on the alignment of the allocation and where `libdiscoverADI.so` places the allocation in the cache line.
- There is a slight chance that binaries compiled with `-xipo=2` might have a memory-optimized code that manipulates addresses in a way that will lead to false positive SSM errors and as a result also lead to performance degradation due to trap handling.

Example of Using `discover` ADI Mode

This section provides a code sample with Array-out-of-bounds errors, which are then caught and reported by `discover` using ADI mode.

Assume the following sample code resides in a file named `testcode.c`.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

char *arr1 = (char*)malloc(sizeof(char)*STRSZ);
char *arr2 = (char*)malloc(sizeof(char)*STRSZ);
// Buffer overflow due to using "<=" instead of "<"
for (int i=0; i <= STRSZ; i++)
arr1[i] = arr2[i]; // ABR/ABW

free(arr1);
```

```
free(arr2);

char *arr3 = (char*)malloc(sizeof(char)*STRSZ);

arr3[0] = arr2[0];  // FMR
arr2[0] = arr3[3];  // FMWarr3[1] = arr1[1];  // Possible stale pointer/FMR

free(arr2);   // Double Free

return 0;
}
```

You would build the test code with the following command:

```
$ cc testcode.c -g -m64
```

To execute this sample application with ADI mode, use the following command:

```
$ discover -w - -i adi -o a.out.adi a.out
$ ./a.out.adi
```

This command generates the following output, in a `discover` report. For more information about reading and understanding these reports, see “Analyzing `discover` Reports” on page 35.

```
ERROR 1 (ABR): reading memory beyond array bounds at address 0x3fffffff7d47e080 {memory:
 v8}:
              main() + 0x48  <a.c:13>
                     10:
                     11:    // Buffer overflow due to using "<=" instead of "<"
                     12:    for (int i=0; i <= STRSZ; i++)
                     13:=>    arr1[i] = arr2[i]; // ABR/ABW
                     14:
                     15:    free(arr1);
                     16:    free(arr2);
   was allocated at (0x3fffffff7d47e040, 64 bytes):
              main() + 0x1c  <a.c:9>
                      6:    {
                      7:
                      8:    char *arr1 = (char*)malloc(sizeof(char)*STRSZ);
                      9:=>  char *arr2 = (char*)malloc(sizeof(char)*STRSZ);
                     10:
                     11:      // Buffer overflow due to using "<=" instead of "<"
                     12:      for (int i=0; i <= STRSZ; i++)
ERROR 2 (ABW): writing to memory beyond array bounds at address 0x2fffffff7d47e040
 {memory: v3}:
              main() + 0x54  <a.c:13>
                      10:
                      11:      // Buffer overflow due to using "<=" instead of "<"
```

```
                    12:      for (int i=0; i <= STRSZ; i++)
                    13:=>       arr1[i] = arr2[i]; // ABR/ABW
                    14:
                    15:      free(arr1);
                    16:      free(arr2);
    was allocated at (0x2fffffff7d47e000, 64 bytes):
              main() + 0x8  <a.c:8>
                    5:    int main()
                    6:    {
                    7:
                    8:=>    char *arr1 = (char*)malloc(sizeof(char)*STRSZ);
                    9:      char *arr2 =(char*)malloc(sizeof(char)*STRSZ);
                    10:
                    11:      // Buffer overflow due to using "<=" instead of "<"
ERROR 3 (FMR): reading from freed memory at address 0x3fffffff7d47e040 {memory: v10}:
              main() + 0xa0  <a.c:20>
                    17:
                    18:      char *arr3 = (char*)malloc(sizeof(char)*STRSZ);
                    19:
                    20:=>    arr3[0] = arr2[0];  // FMR
                    21:      arr2[0] = arr3[3];  // FMW
                    22:      arr3[1] = arr1[1];  // Possible stale pointer/FMR
                    23:
     was allocated at (0x3fffffff7d47e040, 64 bytes):
              main() + 0x1c <a.c:9>
                     6:    {
                     7:
                     8:      char *arr1 = (char*)malloc(sizeof(char)*STRSZ);
                     9:=>    char *arr2 = (char*)malloc(sizeof(char)*STRSZ);
                    10:
                    11:      // Buffer overflow due to using "<=" instead of "<"
                    12:      for (int i=0; i <= STRSZ; i++)
         freed at (0x3fffffff7d47e040, 64 bytes):
              main() + 0x80 <a.c:16>
                    13:         arr1[i] = arr2[i]; // ABR/ABW
                    14:
                    15:      free(arr1);
                    16:=>    free(arr2);
                    17:
                    18:      char *arr3 = (char*)malloc(sizeof(char)*STRSZ);
                    19:
ERROR 4 (FMW): writing to freed memory at address 0x3fffffff7d47e040 {memory: v10}:
              main() + 0xb8 <a.c:21>
                    18:      char *arr3 = (char*)malloc(sizeof(char)*STRSZ);
                    19:
                    20:      arr3[0] = arr2[0];  // FMR
                    21:=>    arr2[0] = arr3[3];  // FMW
```

```
                        22:      arr3[1] = arr1[1];  // Possible stalepointer/FMR

                        23:
                        24:      free(arr2);   // Double Free
               was allocated at (0x3fffffff7d47e040, 64 bytes):
                      main() + 0x1c <a.c:9>
                               6:    {
                               7:
                               8:      char *arr1 = (char*)malloc(sizeof(char)*STRSZ);
                               9:=>    char *arr2 =(char*)malloc(sizeof(char)
*STRSZ);
                        10:
                        11:      // Buffer overflow due to using "<=" instead of "<"
                        12:      for (int i=0; i <= STRSZ; i++)
               freed at (0x3fffffff7d47e040, 64 bytes):
                      main() + 0x80  <a.c:16>
                        13:         arr1[i] = arr2[i]; // ABR/ABW
                        14:
                        15:      free(arr1);
                        16:=>    free(arr2);
                        17:
                        18:      char *arr3 = (char*)malloc(sizeof(char)*STRSZ);
                        19:
ERROR 5 (FMR): reading from freed memory at address 0x2fffffff7d47e001 {memory: v3}:
                      main() + 0xc0 <a.c:22>
                        19:
                        20:      arr3[0] = arr2[0];  // FMR
                        21:      arr2[0] = arr3[3];  // FMW
                        22:=>    arr3[1] = arr1[1];  // Possible stale pointer/FMR
                        23:
                        24:      free(arr2);   // Double Free
                        25:
               was allocated at (0x2fffffff7d47e000, 64 bytes):
                      main() + 0x8  <a.c:8>
                               5:    int main()
                               6:    {
                               7:
                               8:=>    char *arr1 = (char*)malloc(sizeof(char)*STRSZ);
                               9:      char *arr2 = (char*)malloc(sizeof(char)*STRSZ);
                        10:
                        11:      // Buffer overflow due to using "<=" instead of "<"
               freed at (0x2fffffff7d47e000, 64 bytes):
                     main() + 0x74 <a.c:15>
                        12:      for (int i=0; i <= STRSZ; i++)
                        13:      arr1[i] = arr2[i]; //ABR/ABW
                        14:
                        15:=>    free(arr1);
                        16:      free(arr2);
```

```
                    17:
                    18:      char *arr3 = (char*)malloc(sizeof(char)*STRSZ);
ERROR 6 (DFM):  double freeing memory at address 0x3fffffff7d47e040 {memory: v10}:
              main() + 0xd0  <a.c:24>
                    21:     arr2[0] = arr3[3];  // FMW
                    22:     arr3[1] = arr1[1];  // Possible stale pointer/FMR
                    23:
                    24:=>   free(arr2);   // Double Free
                    25:
                    26:      return 0;
                    27:    }
        was allocated at (0x3fffffff7d47e040, 64 bytes):
              main() + 0x1c  <a.c:9>
                     6:    {
                     7:
                     8:      char *arr1 = (char*)malloc(sizeof(char)*STRSZ);

                     9:=>    char *arr2 = (char*)malloc(sizeof(char)*STRSZ);
                    10:
                    11:      // Buffer overflow due to using "<=" instead of "<"
                    12:      for (int i=0; i <= STRSZ; i++)
        freed at (0x3fffffff7d47e040, 64 bytes):
               main() + 0x80 <a.c:16>
                    13:         arr1[i] = arr2[i]; // ABR/ABW
                    14:
                    15:      free(arr1);
                    16:=>    free(arr2);
                    17:
                    18:      char *arr3 = (char*)malloc(sizeof(char)*STRSZ);
                    19:
DISCOVER SUMMARY:
              unique errors   : 6 (6 total)
```

Analyzing `discover` Reports

The `discover` report provides you with information to effectively pinpoint and fix the problems in your source code.

By default, the report is written in HTML format to *output-file*`.html`, where *output-file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

When you instrument your binary, you can use the `-H` option to request that the HTML output be written to a specified file, or the `-w` option to request that it be written to a text file.

After your binary is instrumented, if you want to write the report to a different file for a subsequent run of the program, you can change the settings of the -H and -w options for the report through the SUNW_DISCOVER_OPTIONS environment variable. For more information, see "SUNW_DISCOVER_OPTIONS Environment Variable" on page 50.

Note - If you specify the -a option while instrumenting your code, you must use Code Analyzer or the codean command to read the report.

Analyzing the HTML Report

The HTML report format provides interactive analysis of your program. The data in HTML format can easily be shared between developers using email or placement on a web page. Combined with JavaScript interactive features, this format provides a convenient way to navigate through the discover messages.

This section describes the HTML report, which includes the following tabs:

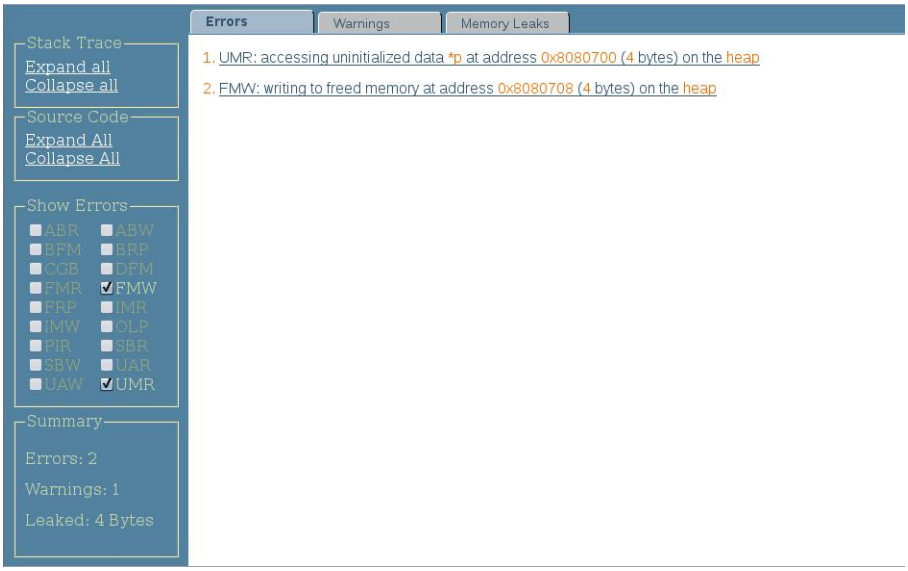
- "Using the Errors Tab" on page 36
- "Using the Warnings Tab" on page 38
- "Using the Memory Leaks Tab" on page 39

The Errors tab, Warnings tab , and Memory Leaks tab let you navigate through error messages, warning messages, and the memory leak report, respectively.

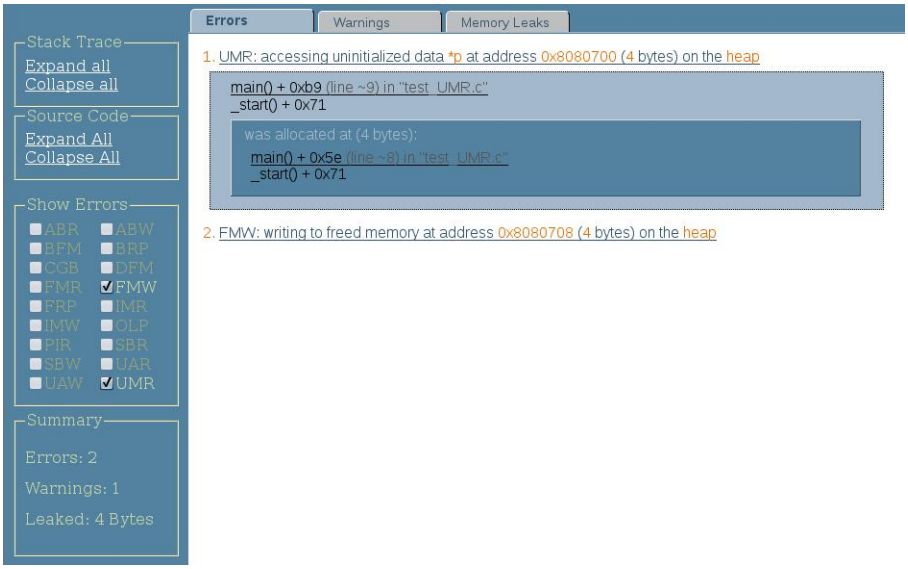
The control panel on the left enables you to change the contents of the tab that is currently displayed on the right. See "Using the Control Panel" on page 41.

Using the Errors Tab

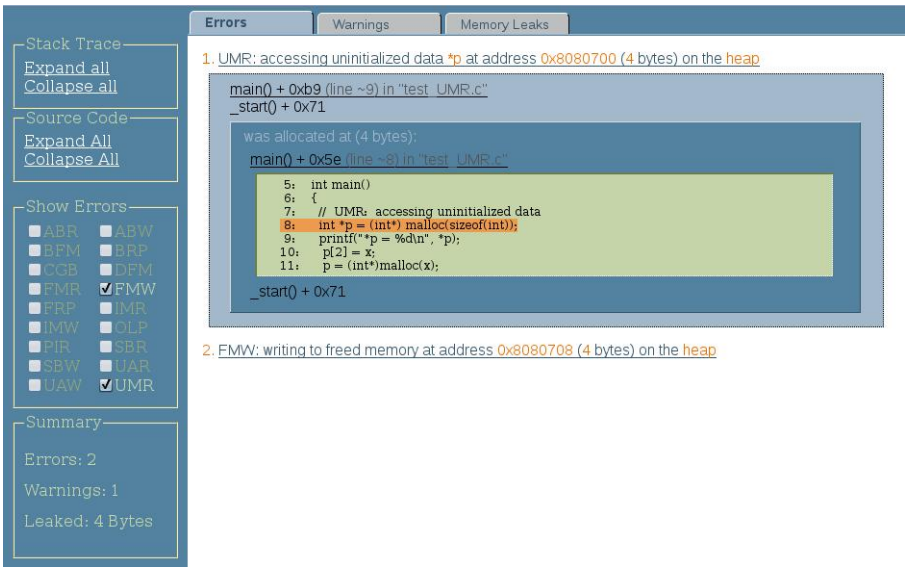
When you first open an HTML report in your browser, the Errors tab is selected and displays the list of memory access errors that occurred during execution of your instrumented binary:



When you click an error, the stack trace at the time of the error is displayed:

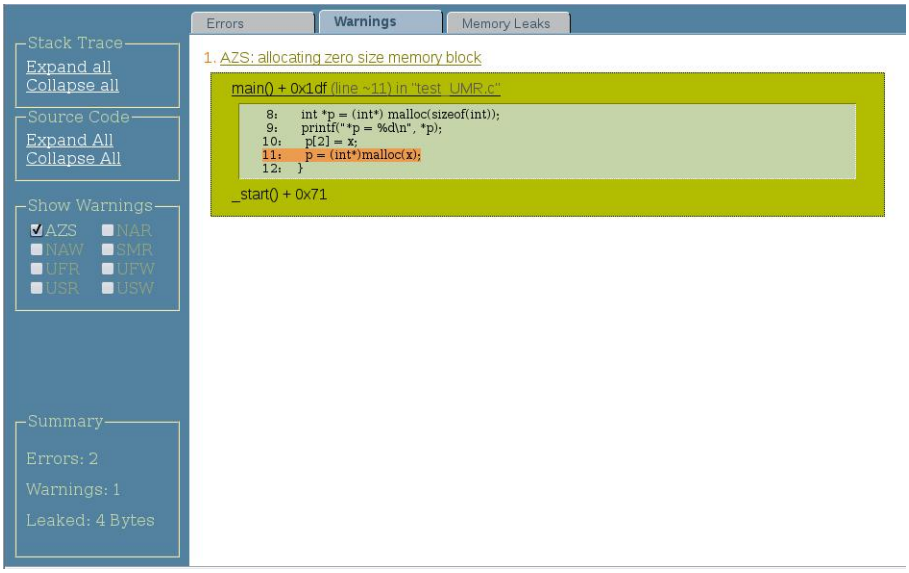


If you compiled your code with the -g option, you can see the source code for each function in the stack trace by clicking the function:



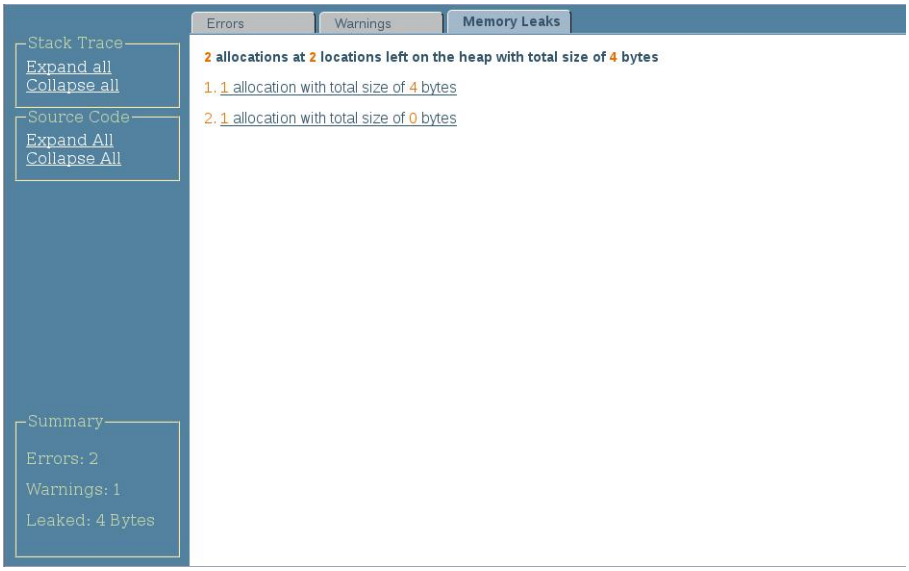
Using the Warnings Tab

The Warnings tab displays all of the warning messages for possible access errors. When you click a warning, the stack trace at the time of the warning is displayed. If you compiled your code with the -g option, you can see the source code for each function in the stack trace by clicking the function:

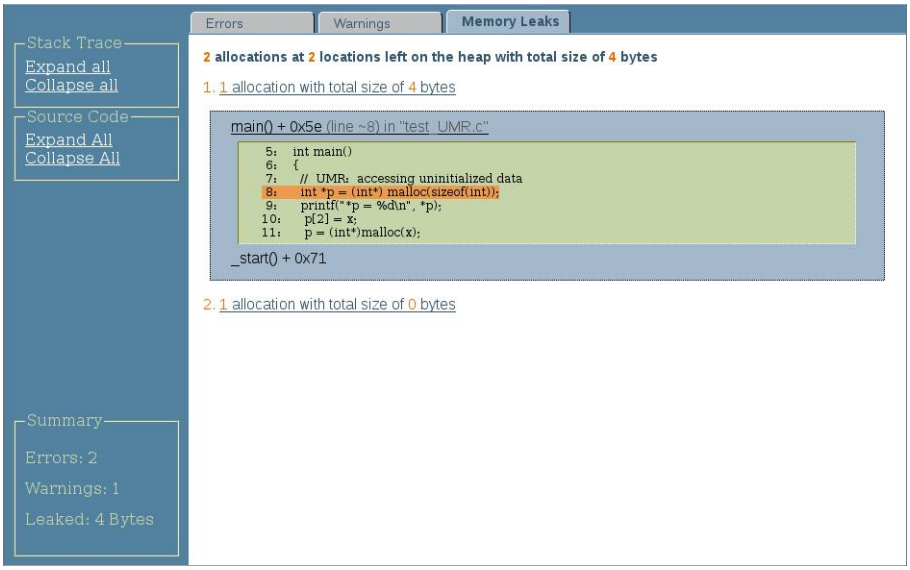


Using the Memory Leaks Tab

The Memory Leaks tab displays the total number of blocks remaining allocated at the end of the program's run at the top, with the blocks listed below.:



When you click a block, the stack trace for the block is displayed. If you compiled your code with the -g option, you can see the source code for each function in the stack trace by clicking the function:



Using the Control Panel

To see the stack traces for all of the errors, warnings, and memory leaks, click Expand All in the Stack Traces section of the control panel. To see the source code for all of the functions, click Expand All in the Source Code section of the control panel.

To hide the stack traces or source code for all of the errors, warnings, and memory leaks, click the corresponding Collapse All.

The Show Errors or Show Warnings sections of the control panel is displayed when the relevant tab is selected. By default, the options for all of the detected errors or warnings are checked. To hide a type of error or warning, deselect it.

A summary of the report listing the total numbers of errors and warnings, and the amount of leaked memory, is displayed at the bottom of the control panel.

Analyzing the ASCII Report

The ASCII (text) format of the `discover` report is suitable for processing by scripts or when you do not have access to a web browser. The following example shows a sample ASCII report.

```
$ a.out

ERROR 1 (UAW): writing to unallocated memory at address 0x50088 (4 bytes) at:
main() + 0x2a0  <ui.c:20>
17:     t = malloc(32);
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:=>    t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:     foo();
23:     bar();
ERROR 2 (UMR): accessing uninitialized data from address 0x50010 (4 bytes) at:
main() + 0x16c  <ui.c:21>$
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
21:=>   printf("%d\n", t[2]);   //UMR
22:     foo();
23:     bar();
24:    }
was allocated at (32 bytes):
main() + 0x24  <ui.c:17>
14:     x = (int*)malloc(size); // AZS warning
```

```
15:    }
16:    int main() {
17:=>   t = malloc(32);
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
0
WARNING 1 (AZS): allocating zero size memory block at:
foo() + 0xf4  <ui.c:14>
11:    void foo() {
12:     x = malloc(128);
13:     free(x);
14:=>   x = (int*)malloc(size); // AZS warning
15:    }
16:    int main() {
17:     t = malloc(32);
main() + 0x18c  <ui.c:22>
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:=>   foo();
23:     bar();
24:    }

***************** Discover Memory Report *****************

1 block at 1 location left allocated on heap with a total size of 128 bytes

1 block with total size of 128 bytes
bar() + 0x24  <ui.c:9>
6:           7:    void bar() {
8:     int *y;
9:=>   y = malloc(128);  // Memory leak
10:    }
11:    void foo() {
12:     x = malloc(128);
main() + 0x194  <ui.c:23>
20:      t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:     foo();
23:=>   bar();
24:    }

ERROR 1: repeats 100 times
DISCOVER SUMMARY:
unique errors   : 2 (101 total, 0 filtered)
unique warnings : 1 (1 total, 0 filtered)
```

The report consists of error and warning messages followed by a summary.

ASCII Warning and Error Message Descriptions

The error message starts with the word ERROR and contains a three-letter code, an ID number, and an error description (writing to unallocated memory in the example). Other details include the memory address that was accessed and the number of bytes read or written. Following the description is a stack trace at the time of the error that pinpoints the location of the error in the process life cycle.

If you compiled the program with the -g option, the stack trace includes the source file name and line number. If the source file is accessible, the source code in the vicinity of the error is printed. The target source line in each frame is indicated by the ⇒ symbol.

When the same kind of error at the same memory location with the same number of bytes repeats, the complete message including the stack trace is printed only once. Subsequent occurrences of the error are counted and a repetition count, as shown in the following example, is listed at the end of the report for each identical error that occurs multiple times.

```
ERROR 1: repeats 100 times
```

If the address of the faulty memory access is on the heap, then information on the corresponding heap block is printed after the stack trace. The information includes the block starting address and size, and a stack trace at the time the block was allocated. If the block was freed, the report includes a stack trace of the deallocation point.

Warning messages appear in the same format as error messages except that they start with the word WARNING. In general, these messages alert you to conditions that do not affect application functionality but provide useful information that you can use to improve the program. For example, allocating memory of zero size is not harmful but if it happens too often, it can potentially degrade performance.

ASCII Memory Leak Report

The memory leak report contains information about memory blocks allocated on the heap but not released at program exit. The following example shows a sample memory leak report.

```
$ DISCOVER_MEMORY_LEAKS=1 ./a.out
...
***************** Discover Memory Report *****************
```

```
2 blocks left allocated on heap with total size of 44 bytes
block at 0x50008 (40 bytes long) was allocated at:
malloc() + 0x168 [libdiscoverADI.so:0xea54]
f() + 0x1c [a.out:0x3001c]
<discover_example.c:9>:
8:    {
9:=>    int *a = (int *)malloc( n * sizeof(int) );
10:      int i, j, k;
main() + 0x1c [a.out:0x304a8]
<discover_example.c:33>:
32:      /* Print first N=10 Fibonacci numbers */
33:=>    a = f(N);
34:      printf("First %d Fibonacci numbers:\n", N);
...
```

The first line following the header summarizes the number of heap blocks left allocated on the heap and their total size. The reported size is from the developer's perspective, that is, it does not include the bookkeeping overhead of the memory allocator.

ASCII Stack Trace Report

After the memory leak summary, detailed information is provided on each unfreed heap block with a stack trace of its allocation point. The stack trace report is similar to the one described for error and warning messages.

ASCII Report Summary

The `discover` report concludes with an overall summary. It reports the number of unique warnings and errors and, in parentheses, the total numbers of errors and warnings, including repeated ones. For example:

```
DISCOVER SUMMARY:
unique errors   : 3 (3 total)
unique warnings : 1 (5 total)
```

`discover` APIs and Environment Variables

There are several `discover` APIs and environment variables that you can specify in your code.

discover APIs

Oracle Developer Studio 12.5 implements six new discover functions that you can call from your program to receive memory leak and memory allocation information. These functions print the information on stderr. At the end of the program output, discover by default prints the final memory report with the memory leaks in the program. To use these APIs, the source file of the application needs to include the header file for discover: #include <discoverAPI.h>.

The functions and what they report are as follows:

discover_report_all_inuse()
: Reports all memory allocations

discover_report_unreported_inuse()
: Reports all memory allocations not previously reported.

discover_mark_all_inuse_as_reported()
: Marks all memory allocations thus far as reported.

discover_report_all_leaks()
: Reports all memory leaks.

discover_report_unreported_leaks()
: Reports all memory leaks not previously reported.

discover_mark_all_leaks_as_reported()
: Marks all memory leaks thus far as reported

This section describes some methods for working with discover APIs.

Note - The discover APIs will not work with ADI mode.

Finding Memory Leaks With discover APIs

For each function specified in your code, discover reports the stack of where the memory was allocated. Memory leaks are allocated memory that is unreachable in the program.

The following example shows how to use these APIs:

```
$ cat -n tdata.C
```

```
     1    #include <discoverAPI.h>
     2
     3    void foo()
     4    {
     5      int *j = new int;
     6    }
     7
     8    int main()
     9    {
    10      foo();
    11      discover_report_all_leaks();
    12
    13      foo();
    14      discover_report_unreported_leaks();
    15
    16      return 0;
    17    }
$ CC -g tdata.C
$ discover -w - a.out
$ a.out
```

The following example shows the expected output.

```
******** discover_report_all_leaks() Report ********
1 allocation at 1 location left on the heap with a total size of 4 bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main()+0x1a  <tdata.C:10>
9:    {
10:=>    foo();
11:      discover_report_all_leaks();
12:
13:      foo();

************************************************************
******** discover_report_unreported_leaks() Report ********
1 allocation at 1 location left on the heap with a total size of 4 bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
```

```
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:int main()
main() + 0x24  <tdata.C:13>
10:      foo();
11:      discover_report_all_leaks();
12:
13:=>    foo();
14:      discover_report_unreported_leaks();
15:
16:return 0;

************************************************************
***************** Discover Memory Report *****************
2 allocations at 2 locations left on the heap with a total size of 8     bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main() +    0x1a  <tdata.C:10>
7:
8:    int main()
9:    {             10:=>    foo();
11:      discover_report_all_leaks();
12:
13:      foo();
LEAK 2: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main() + 0x24  <tdata.C:13>
10:      foo();
```

```
11:      discover_report_all_leaks();
12:
13:=>    foo();
14:         discover_report_unreported_leaks();
15:
16:      return 0;

DISCOVER SUMMARY:
unique errors   : 0 (0 total)
unique warnings : 0 (0 total)
```

Finding Leaks in a Server or Long-Running Program

If you have a long-running program or a server that never exits, you can call these `discover` functions using `dbx` at any time, even if you have not put the calls in your code. The program must have been run with at least the lite mode of `discover` using the `-l` option. Note that `dbx` can attach to a running program. The following example shows how to find leaks in a long-running program.

EXAMPLE 2 Finding Two Leaks in a Long Running Program

For this example, the `a.out` file is a long-running program with two processes, each with one leak. Each process is assigned a process ID.

The following `rl` script contains the commands to ask the program to report unreported memory leaks.

```
#!/bin/sh
dbx - $1 > /dev/null 2> &1 << END
call discover_report_unreported_leaks()
exit
END
```

Once you have a program and a script, you can use `discover` and run the program.

```
% discover -l -w - a.out
% a.out
8252: Parent allocation 64
8253: Child allocation 32
```

In a separate terminal window, you can run the script on the parent process.

```
% rl 8252
```

The program reports the following information for the parent process:

```
******** discover_report_unreported_leaks() Report ********

1 allocation at 1 location left on the heap with a total size of 64 bytes

    LEAK 1: 1 allocation with total size of 64 bytes
    main() + 0x1e  <xx.c:17>
        14:
        15:      if (child > 0) {
        16:
        17:=>      void *p = malloc(64);
        18:        printf("%jd: Parent allocation 64\n", (intmax_t)getpid());
        19:        p = 0;
        20:        for (int j=0; j < 1000; j++) sleep(1);


**********************************************************
```

Run the script again for the child process.

```
% rl 8253
```

The program reports the following information for the child process:

```
******** discover_report_unreported_leaks() Report ********

1 allocation at 1 location left on the heap with a total size of 32 bytes

    LEAK 1: 1 allocation with total size of 32 bytes
    main() + 0x80  <xx.c:24>
        21:      }
        22:
        23:      else {
        24:=>      void *p = malloc(32);
        25:        printf("%jd: Child allocation 32\n", (intmax_t)getpid());
        26:        p = 0;
        27:        for (int j=0; j < 1000; j++) sleep(1);


**********************************************************
```

You can use the script repeatedly to find any new leaks.

SUNW_DISCOVER_OPTIONS Environment Variable

You can change the runtime behavior of an instrumented binary by setting the SUNW_DISCOVER_OPTIONS environment variable to a list of the command-line options -a, -A, -b, -e, -E, -f, -F, -H, -l, -L, -m, -P, -S, and -w. For example, if you want to change the number of errors reported to 50 and limit the stack depth in the report to 3, you would set the environment variable to the following:

```
-e 50 -S 3
```

Memory Access Errors and Warnings

The discover utility detects and reports many memory access errors, as well as warning you about accesses that might be errors.

Memory Access Errors

discover detects the following memory access errors:

- ABR: beyond array bounds read
- ABW: beyond array bounds write
- BFM: bad free memory
- BRP: bad reallocate address parameter
- CGB: corrupted array guard block
- DFM: double freeing memory
- FMR: freed memory read
- FMW: freed memory write
- FRP: freed realloc parameter
- IMR: invalid memory read
- IMW: invalid memory write
- Memory leak
- OLP: overlapping source and destination
- PIR: partially initialized read
- SBR: beyond stack frame bounds read
- SBW: beyond stack frame bounds write
- UAR: unallocated memory read

- UAW: unallocated memory write
- UMR: uninitialized memory read

The following sections list some simple sample programs that will produce some of these errors.

Beyond Array Bounds Read (ABR)

Example:

```
  //  ABR: reading memory beyond array bounds at address 0x%1x (%d byte%s)
int *a = (int*) malloc(sizeof(int[5]));
printf("a[5] = %d\n",a[5]);
```

The `discover` utility also detects static-type ABR errors.

```
    int globalarray[5];

   int main(){
     int i, j;
     for(i = 0; i < 7; i++) {
       j = globalarray[i-1];   // Reading memory beyond static/global array bounds
     }
     return 0;
   }
```

Beyond Array Bounds Write (ABW)

Example:

```
  //  ABW:  writing to memory beyond array bounds
int *a = (int*) malloc(sizeof(int[5]));
a[5] = 5;
```

The `discover` utility also detects static-type ABW errors.

```
int globalarray[5];

   int main(){
     int i;
     for(i = 0; i < 7; i++) {
       globalarray[i-1] = i;   // Writing to memory beyond static/global array bounds
     }
     return 0;
   }
```

Bad Free Memory (BFM)

Example:

```
  //  BFM:  freeing wrong memory block
int *p = (int*) malloc(sizeof(int));
free(p+1);
```

Bad Realloc Address Parameter (BRP)

Example:

```
  //  BRP:  bad address parameter for realloc 0x%1x
int *p = (int*) realloc(0,sizeof(int));
int *q = (int*) realloc(p+20,sizeof(int[2]));
```

Corrupted Guard Block (CGB)

Example:

```
  // CGB: writing past the end of a dynamically allocated array, or being in the "red
 zone".
    #include <stdio.h>
   #include <stdlib.h>

   int main() {
     int *p = (int *) malloc(sizeof(int)*4);
     *(p+5) = 10; //  Corrupted array guard block detected (only when the code is not
 annotated)
     free(p);

     return 0;
   }
```

Double Freeing Memory (DFM)

Example:

```
  // DFM: double freeing memory
int *p = (int*) malloc(sizeof(int));
free(p);
free(p);'
```

Freed Memory Read (FMR)

Example:

```
  //  FMR: reading from freed memory at address 0x%1x  (%d byte%s)
int *p = (int*) malloc(sizeof(int));
free(p);
printf("p = 0x%h\n",*p);
```

Freed Memory Write (FMW)

Example:

```
  //  FMW: writing to freed memory at address 0x%1x (%d byte%s)
int *p = (int*) malloc(sizeof(int));
free(p);
*p = 1;
```

Freed Realloc Parameter (FRP)

Example:

```
  //  FRP:  freed pointer passed to realloc
int *p = (int*) malloc(sizeof(int));
free(0);
int *q = (int*) realloc(p,sizeof(int[2]));
```

Invalid Memory Read (IMR)

Example:

```
  //  IMR:  read from invalid memory address
int *p = 0;
int i = *p;   // generates Signal 11...
```

Invalid Memory Write (IMW)

Example:

```
  //  IMW:  write to invalid memory address
int *p = 0;
```

```
*p = 1;       // generates Signal 11...
```

Memory Leak

Example:

```
  //  Memory Leak: memory allocated but not freed before exit or escaping from the
 function
       int foo()
   {
     int *p = (int*) malloc(sizeof(int));
     if (x) {
      p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
     }
   }                                      // The 2nd malloc leaked here
```

Overlapping Source and Destination (OLP)

Example:

```
  //  OLP: source and destination overlap
char *s=(char *) malloc(15);
memset(s, 'x', 15);
memcpy(s, s+5, 10);
return 0;
```

Partially Initialized Read (PIR)

Example:

```
  //  PIR:  accessing partially initialized data
int *p = (int*) malloc(sizeof(int));
*((char*)p) = 'c';
printf("*(p = %d\n",*(p+1));
```

Beyond Stack Bounds Read (SBR)

Example:

```
  //  SBR: reading beyond stack frame bounds
int a[2]={0,1};
printf("a[-10]=%d\n",a[-10]);
```

```
return 0;
```

Beyond Stack Bounds Write (SBW)

Example:

```
  //  SBW: writing beyond stack frame bounds
int a[2]={0,1)'
a[-10]=2;
return 0;
```

Unallocated Memory Read (UAR)

Example:

```
  //  UAR" reading from unallocated memory
int *p = (int*) malloc(sizeof(int));
printf("*(p+1) = %d\n",*(p+1));
```

Unallocated Memory Write (UAW)

Example:

```
  // UMR: accessing uninitialized data from address 0x%1x (A%d byte%s)
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n",*p);
```

Memory Access Warnings

The `discover` utility reports the following memory access warnings:

- AZS: allocating zero size
- SMR: speculative uninitialized memory read

The following sections gives examples of these warnings.

Allocating Zero Size (AZS)

Example:

```
#include <stdlib>
int main()
{
  int *p = malloc(); // Allocating zero size memory block
}
```

Memory Leak (MLK)

Possible causes: Memory is allocated but not freed before exit or escaping from the function.

Example:

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
 }
}                                     // The 2nd malloc leaked here
```

Speculative Memory Read (SMR)

```
int i;
if (foo(&i) != 0)  /* foo returns nonzero if it has initialized i */
printf("5d\n", i);
```

The compiler might generate the following equivalent code for the above source:

```
int i;
int t1, t2'
t1 = foo(&i);
t2 = i; /* value in i is loaded. So even if t1 is 0, we have uninitialized read due to
 speculative load */
if (t1 != 0)
printf("%d\n", t2);
```

Interpreting discover Error Messages

In some cases, discover might report an error that is not actually an error. Such cases are called false positives. The discover utility analyzes code at instrumentation time to reduce the occurrence of false positives compared to similar tools, but they might still occur in some

instances. This section provides a few tips that might help you to identify and possibly avoid false positives in discover reports.

Partially Initialized Memory

You can use bit fields in C and C++ to create compact data types. For example:

```
struct my_struct {
unsigned int valid : 1;
char         c;
};
```

In the example, the structure member my_struct.valid takes only one bit in memory. However, on SPARC platforms, the CPU can modify memory only in bytes, so the whole byte containing struct.valid must be loaded in order to access or modify the structure member. Moreover, sometimes the compiler might load several bytes (for example, a machine word of four bytes) at once. When discover detects such a load without additional information, it assumes that all four bytes are used. If, for example, the field my_struct.valid was initialized but the field my_struct.c was not and the machine word containing both fields was loaded, discover would flag a partially initialized memory read (PIR).

Another source of false positives is initialization of a bit field. To write a part of a byte, the compiler must first generate code that loads the byte. If the byte was not written prior to a read, the result is an uninitialized memory read error (UMR).

To avoid false positives for bit fields, use the -g option or the -g0 option when compiling. These options provide extra debugging information to discover to help it identify bit field loads and initialization, which will eliminate most false positives. If you cannot compile with the -g option for some reason, then initialize structures with a function such as memset(). For example:

```
...
struct my_struct s;
/* Initialize structure prior to use */
memset(&sm 0, sizeof(struct my_struct));
...
```

Speculative Loads

Sometimes the compiler generates a load from a known memory address under conditions where the result of the load is not valid on all program paths. This situation often occurs on

SPARC platforms because such a load instruction can be placed in the delay slot of a branch instruction. For example, consider this C code fragment:

```
int i'
if (foo(&i) != 0) { /* foo returns nonzero if it has initialized i */
printf("5d\n", i);
}
```

From this code, the compiler could generate code equivalent to the following example:

```
int i;
int t1, t2'
t1 = foo(&i);
t2 = i; /* value in i is loaded */
if (t1 != 0) {
printf("%d\n", t2);
}
```

Assume that in the example, the function `foo()` returns `0` and does not initialize `i`. The load from `i` is still generated, even though it is not used. However, because `discover` will see the load, it will report a load of an uninitialized variable (UMR).

The `discover` utility uses dataflow analysis to identify such cases whenever possible, but sometimes they are impossible to detect.

You can reduce the occurrence of these types of false positives by compiling with a lower optimization level.

Uninstrumented Code

Sometimes `discover` cannot instrument 100% of your program, especially if some of your code comes from an assembly language source file or a third–party library that cannot be recompiled and so cannot be instrumented. In some cases, `discover` cannot detect the memory blocks the non-instrumented code is accessing and modifying. Assume for example that a function from a third-party shared library initializes a block of memory that is later read by the main (instrumented) program. If `discover` cannot detect that the memory has been initialized by the library, the subsequent read generates an uninitialized memory error (UMR).

To provide a solution for such cases, the `discover` API includes the following functions:

```
void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
void __ped_memory_copy(unsigned long src, unsigned lond dst, long size, unsigned long
 pc);
```

You can call the API functions from your program to notify discover of specific events such as a write to a memory area (__ped_memory_write()) or a read from a memory area (__ped_memory read()). In both cases, the starting address of the memory area is passed in the addr parameter and its size is passed in the size parameter. Set the pc parameter to 0.

Use the __ped_memory_copy function to notify discover of memory that is being copied from one location to another. The starting address of the source memory is passed in the src parameter, the starting address of the destination area is passed in the dst parameter, and the size is passed in the size parameter. Set the pc parameter to 0.

To use the API, declare these functions in your program as weak. For example, include the following code fragment in your source code.

```
#ifdef __cplusplus
extern "C" {
#endif

extern void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_copy(unsigned long src, unsigned long dst, long size, unsigned
 long pc);

#prgama weak __ped_memory_write
#pragma weak __ped_memory_read
#pragma weak __ped_memory_copy

#ifdef __cplusplus
}
#endif
```

The internal discover library, which is linked with your program at instrumentation time, defines the API functions. However, when your program is not instrumented, this library is not linked and thus all calls to the API functions will cause the application to hang. Therefore, you must disable these functions when you are not running your program under discover. Alternatively, you can create a dynamic library with empty definitions of the API functions and link it with your program. In this case, when you run your program without discover your library will be used, but when you run it under discover the real API functions will be called automatically.

Limitations When Using discover

This section describes some known limitations when using discover.

Non-Annotated Code Might Cause False Results

The discover utility can only instrument code that has been described in “Instrumenting a Binary” on page 16 . Uninstrumented code might come from assembly language code linked into the binary or from modules compiled with older compilers or operating systems than those listed in that section. If a function is not instrumented, false positive error messages might be emitted, either for the function, its callers, or its callees. Additionally, some errors might not be diagnosed in uninstrumented functions.

The discover utility cannot instrument assembly language modules or functions that contain asm statements or .il templates.

Furthermore, the Oracle Developer Studio 12.5 C++ runtime libraries do not contain annotation data because they were not compiled with an Oracle Developer Studio compiler.

Machine Instruction Might Differ From Source Code

discover operates on machine code. The tool detects errors on machine instructions such as loads and stores, and correlates the errors with the source code. Because some source code statements do not have associated machine instructions, discover might not seem to detect an obvious user error. For example, consider the following C code fragment:

```
int *p = (int *)malloc(sizeof(int));
int i;

i = *p;  /* compiler may not generate code for this statement */
printf("Hello World!\n");

return;
```

Reading a value stored at the address pointed to by p is a potential user error because the memory was not initialized. However, an optimizing compiler will detect that the variable i is not used, so it will not generate the code for the statement reading from memory and assigning to i. In this case, discover will not report uninitialized memory usage (UMR).

Compiler Options Affect the Generated Code

Compiler-generated code is not predictable. Because the code the compiler generates varies depending on the compiler options you use, including the -O*n* optimization options, the errors

reported by discover might also vary. For example, errors reported in code generated at the -O1 optimization level might not apply to code generated at the -O4 optimization level.

Binaries compiled with the -xlinkopt flag are incompatible with discover.

System Libraries Can Affect the Errors Reported

System libraries are preinstalled with the operating system and cannot be recompiled for instrumentation. The discover utility provides support for the common function from the standard C library (libc.so); that is, discover knows what memory is accessed or modified by these functions. However, if your application uses other system libraries, you might see false positives in the discover report. If false positives are reported, you can call the discover API from your code to eliminate them.

Custom Memory Management Can Affect the Accuracy of the Data

The discover utility can track heap memory when it is allocated by standard programming language mechanisms like malloc(), calloc(), free(), operator new(), and operator delete().

If your application uses a custom memory management system with the standard functions (for example, pool allocation management implemented with malloc()), then discover might not guarantee to correctly report leaks or access to freed memory.

The discover utility does not support the following memory allocators:

- Custom heap allocators that use brk(2)() or sbrk(2)() system calls directly
- Standard heap management function linked statically into a binary
- Memory allocated from the user code using mmap(2)() and shmget(2)() system calls

The signaltstack(2)() function is not supported.

♦ ♦ ♦ CHAPTER 3

Code Coverage Tool (uncover)

The Code Coverage Tool (uncover) software measures the code coverage of applications. This chapter provides information about the following topics:

Requirements for Using uncover

The uncover utility works on binaries compiled with at least Sun Studio 12 Update 1, Oracle Solaris Studio 12.2-12.4 or Oracle Developer Studio 12.5 compilers. It works on a SPARC–based or x86–based system running at least one of the following operating systems: Solaris 10 10/08, Oracle Solaris 11, Oracle Enterprise Linux 5.x, or Oracle Enterprise Linux 6.x.

A binary compiled as described includes information that uncover uses to reliably disassemble the binary to instrument it for coverage data collection.

To enable Uncover to use source code level coverage information use the -g option to generate debug information when compiling the binary. If your binary is not compiled with the -g option, Uncover uses only program counter (PC) based coverage information.

The uncover utility works with any binary built with Oracle Developer Studio compilers, but it works best with binaries built with no optimization option. Previous releases of uncover required at least the -O1 optimization level. If you build your binary with an optimization option, uncover results will be better with lower optimization levels (-O1 or -O2). uncover derives source-line level coverage by relating the instructions to line numbers using the debug information generated when the binary is built with the -g option. At optimization

levels -O3 and higher, the compiler might delete some code that might never be executed or is redundant, which might result in no binary instructions for some source code lines. In such cases, no coverage information is reported for those lines. See "Limitations When Using uncover" on page 80 for more information.

Using uncover

Generating coverage information using Uncover is a three-step process:

1. "Instrumenting the Binary" on page 64
2. "Running the Instrumented Binary" on page 65
3. "Generating and Viewing the Coverage Report" on page 65
4. "Coverage for Shared Libraries" on page 67

This section covers the three steps and provides examples of using Uncover.

Instrumenting the Binary

The input binary can be an executable or a shared library. You must instrument each binary that you want to analyze separately.

You instrument the binary with the uncover command. For example, the following command instruments the binary a.out and overwrites the input a.out with the instrumented a.out. It also creates a directory with the suffix .uc (a.out.uc in this case) in which the coverage data will be collected. A copy of the input binary is saved in this directory.

If you have a binary a.out and a shared library mylib.so, instrument them as follows:

```
$ uncover a.out
$ uncover mylib.so
```

You can use the following options when instrumenting your binary:

-c — Enable reporting of execution counts for instructions, blocks, and functions. By default, only information on code that is covered or not covered is reported. Specify this option both when instrumenting your binary and when generating the coverage report.

-d *directory* — Creates the coverage data directory in *directory*. For uncover, this option is especially important because all three phases of its usage require access to the same exact directory. The -d option used in conjunction

with <*full_path_of_coverage_directory*> ensures that different phases of `uncover` look for the coverage directory in the same location. If you do not use the `-d` with full path names, there could be a mismatch in locating the coverage directory, causing other issues such as multiple profile directories with incomplete information.

`-m on | off` — Enables or disables thread-safe profiling when collecting execution counts with the `-c` runtime option. This option must be used in conjunction with the `-c` option or it has no effect. The `-c` option enables thread-safe profile counting by default. The default is set to `on`. To collect profile counts for a single-threaded application, use the `-c -m off` option. This will turn off the thread-safety feature which is unnecessary for single threaded apps and allows a faster profile run.

`-o` *output-binary-file* — Writes the instrumented binary file to the specified file. The default is to overwrite the input binary file with the instrumented file.

If you run the `uncover` command on an input binary that is already instrumented, `uncover` issues an error message that the binary cannot be instrumented because it is already instrumented, and that you can run it to generate coverage data.

Running the Instrumented Binary

After you have instrumented your binary, you can run it normally. Every time you run the instrumented binary, code coverage data is collected in the coverage data directory with the `.uc` suffix that `uncover` created during the instrumentation. Because `uncover` data collection is multi-thread safe and multi-process safe, there is no restriction on the number of simultaneous runs or threads in the process. The coverage data is accumulated over all of the runs and threads.

Generating and Viewing the Coverage Report

To generate a coverage report, run the `uncover` command on the coverage data directory. For example with binary `a.out` and shared library `mylib.so`:

```
$ uncover a.out.uc
$ uncover mylib.so.uc
```

This command generates an Oracle Developer Studio Performance Analyzer experiment directory called *binary-name*`.er` from the coverage data in the `a.out.uc` directory, starts the

Performance Analyzer GUI, and displays the experiment. The presence of an .er.rc file in the current directory or your home directory might affect the way Performance Analyzer displays the experiment. For more information about .er.rc files, see *Oracle Developer Studio 12.5: Performance Analyzer*.

You can generate the report as HTML and view it in your web browser or as ASCII to view in a terminal window. You can also direct the data to a directory where Code Analyzer can analyze and display it.

-a	Write error data to *binary-name*.analyze/coverage directory for use by Code Analyzer.
-c	Enables reporting of execution counts for instructions, blocks, and functions. By default only information on code that is covered or not covered is reported. (Specify this option both when instrumenting your binary and when generating the coverage report.)
-e on | off	Determines whether to generate experiment directory for the coverage report and display the experiment in the Performance Analyzer GUI. The default is on.
-H *html-directory*	Save the coverage data as HTML in the specified directory and automatically display it in your web browser.
-h or -?	Display help.
-n	Generate coverage reports but do not start viewers like Performance Analyzer or web browser.
-t *ascii-file*	Generate an ASCII coverage report in the specified file.
-V	Print uncover version and exit.
-v	Verbose. Print a log of what Uncover is doing.

Only one output format is enabled. If you specify multiple output options, uncover uses the last option in the command.

EXAMPLE 3 uncover Command Examples

```
$ uncover a.out
```

This command instruments the binary a.out, overwrites the input a.out, creates an a.out.uc coverage data directory in the current directory, and saves a copy of the input a.out in the

a.out.uc directory. If a.out is already instrumented, a warning message is displayed and no instrumentation is done.

```
$ uncover mylib.so
```

This command instruments the shared library mylib.so, overwrites the input mylib.so, creates a mylib.so.uc coverage data directory in the current directory, and saves a copy of the input mylib.so in the mylib.so.uc directory. If mylib.so is already instrumented, a warning message is displayed and no instrumentation is performed.

```
$ uncover -d coverage a.out
```

This command creates the a.out.uc coverage directory in the directory coverage.

```
$ uncover a.out.uc
```

This command uses the data in the a.out.uc coverage directory to create a code coverage experiment a.out.er in your working directory, and starts Performance Analyzer to display the experiment.

```
$ uncover mylib.so.uc
```

This command uses the data in the mylib.so.uc coverage directory to create a code coverage experiment mylib.so.er in your working directory, and starts Performance Analyzer to display the experiment.

```
$ uncover -H a.out.html a.out.uc
```

This command uses the data in the a.out.uc coverage directory to create an HTML code coverage report in the directory a.out.html and displays the report in your web browser.

```
$ uncover -t a.out.txt a.out.uc
```

This command uses the data in the a.out.uc coverage directory to create an ASCII code coverage report in the file a.out.txt.

```
$ uncover -a a.out.uc
```

This command uses the data in the a.out.uc coverage directory to create a coverage report in the *binary-name*.analyze/coverage directory for use by Code Analyzer.

Coverage for Shared Libraries

Each binary in the application needs to be instrumented separately. For instance, if the application has an executable a.out and a shared library libfoo.so, you need to instrument each one in order to receive coverage for both.

This command instruments the executable `a.out` and shared library `libfoo.so`.

```
% uncover -d <coverage_dir> a.out
% uncover -d <coverage_dir> libfoo.so
```

This command runs the application to collect coverage data in `<coverage_dir>/a.out.uc` and `<coverage_dir>/libfoo.so.uc`.

```
% ./a.out
```

This command displays the executable `a.out`.

```
% uncover <coverage_dir>/a.out.uc
```

This command views coverage for the shared library `libfoo.so`.

```
% uncover <coverage_dir>/libfoo.so.uc
```

Understanding the Coverage Report in Performance Analyzer

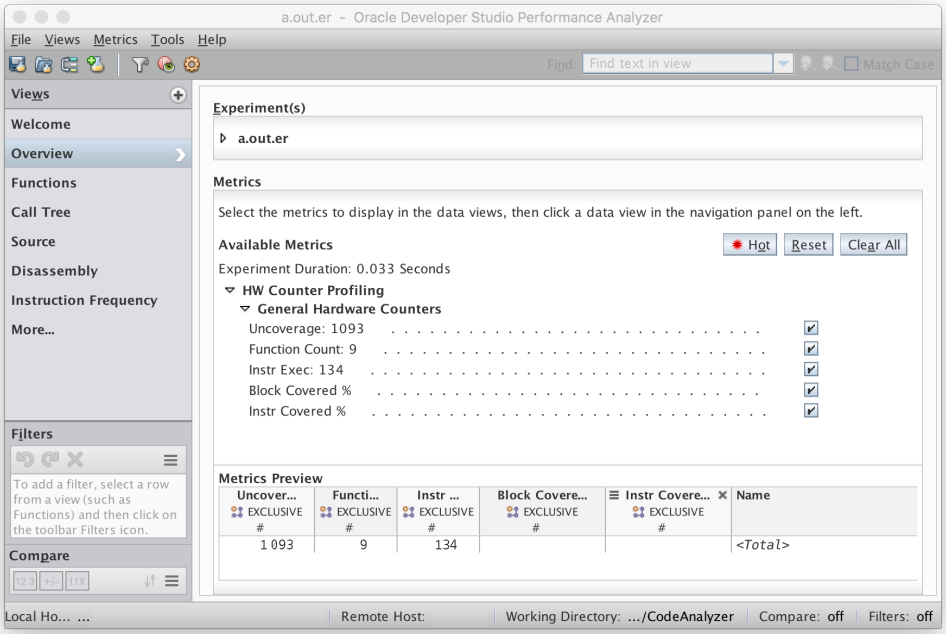
By default, when you run the `uncover` command on the coverage directory, the coverage report opens as an experiment in Oracle Developer Studio Performance Analyzer. This section describes Performance Analyzer interface that displays the coverage data.

For more information about Performance Analyzer, see the integrated help and *Oracle Developer Studio 12.5: Performance Analyzer*.

Overview Screen

When you open the coverage report in Performance Analyzer, the Overview screen is displayed. This view shows the Experiment(s) that you are running, the Metrics of the experiment, and the Metrics Preview.

The following figure shows the Overview screen in Performance Analyzer.



Functions View

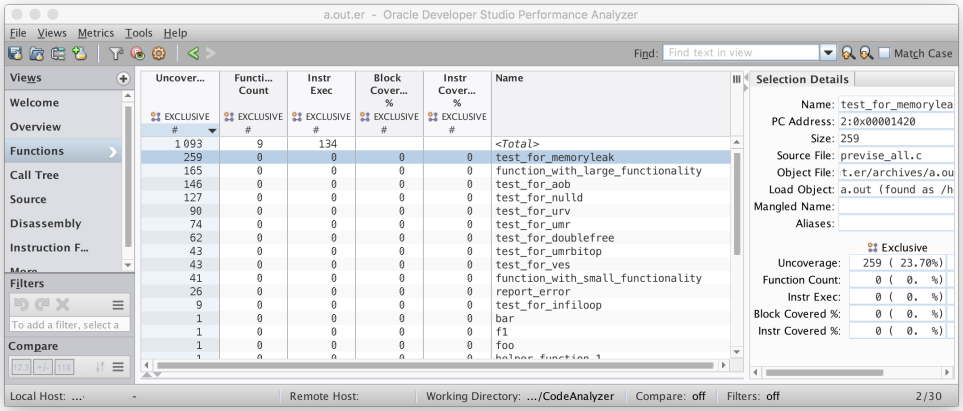
In the navigation panel, click the Functions view to display the program's functions and exclusive metrics. To sort the data according to the value of a particular metric, click the desired column header. Clicking the arrow under the column header reverses the sort order.

The metrics include the following:

Uncoverage	Uncoverage counter, indicating the number of bytes that can be covered for the function.
Function Count	Function counter, indicating which functions are covered.
Instr Exec	Instr-Exec counter, indicating if an instruction was executed in a function.
Block Covered %	Block covered % counter, indicating the percentage of blocks covered in a function.

Instr Covered %
: Instr covered % counter, indicating the percentage of instructions covered in a function.

The following figure shows a coverage report in Performance Analyzer, sorted by Uncoverage.



Uncoverage Counter

The Uncoverage counter is a very powerful feature of `uncover`. If you use this column as the sort key in decreasing order, the top functions in the display are the functions that offer the greatest potential to increase coverage. In the previous figure, the `test_for_memory_leak()` function is at the top of the list because it has the largest number in the Uncoverage column.

The Uncoverage number for the `test_for_memory_leak()` function is number of bytes of code that could potentially be covered if a test is added to the suite that causes the function to be called. The amount that coverage would actually increase varies according to the structure of the function. If no branches are in the function, and all the functions it calls are also straight line functions, then coverage will increase by the stated number of bytes. However, the coverage increase usually is less than the potential, perhaps much less.

The uncovered functions with non-zero values in the Uncoverage column are called root uncovered functions, meaning that they are all called by covered functions. Functions that are called only by non-root uncovered functions do not have their own Uncoverage numbers. It is presumed that these functions will be either covered or revealed as uncovered, in subsequent runs as the test suite is improved to cover the high-potential uncovered functions.

The coverage numbers are non-exclusive.

Function Count

The Function Count column reports the covered functions and uncovered functions. If the count is zero, the function is not covered. If the count is non-zero, the function is covered. If any instruction in the function is executed, the function is considered to be covered.

You can detect non-top-level uncovered functions in this column. If both the Function Count and Uncoverage columns state zero, then the function is not a top-level covered function.

Instr Exec Counter

The Instr Exec counter displays the covered instructions and uncovered instructions. A zero count means that the instruction is not executed; a non-zero count means that the instruction is executed.

In the Functions view, this counter shows the total number of instructions executed for each function. This counter also appears in the Source view and the Disassembly view.

Block Covered % Counter

For each function, the Block Covered % counter, which displays the percentage of basic blocks in the function that are covered. This number indicates how well the function is covered. Disregard this entry in the Total row; it is the sum of percentages in the column and is meaningless.

Instr Covered % Counter

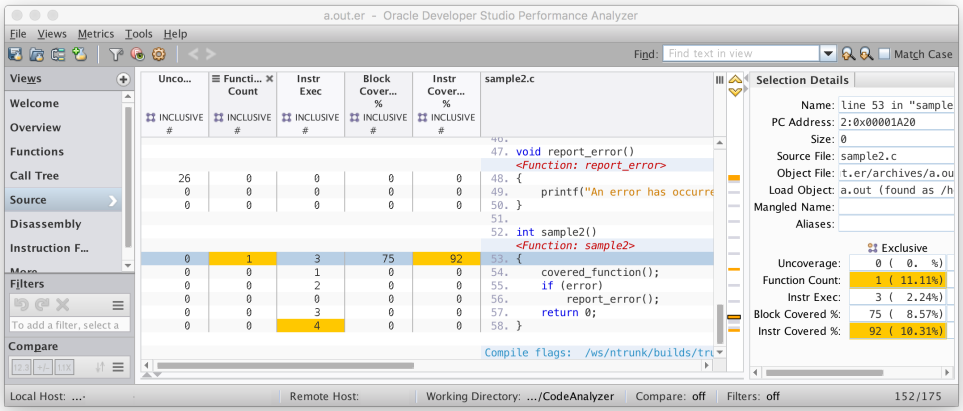
For each function, the Instr Covered % counter displays the percentage of instructions in the function that are covered. This number indicates how well the function is covered. Disregard this entry in the Total row; it is the sum of percentages in the column and is meaningless.

Source View

If you compiled your binary with the `-g` option, the Source view displays the source code of your program. Because `uncover` instruments your program at the binary level and you have compiled the program with optimization, the coverage information in this view can be difficult to interpret.

The Instr Exec counter in the Source view shows the total number of instructions executed for each source line, which is essentially the statement-level code coverage information. A non-zero value implies that the statement is covered; a zero value means that the statement is not covered. Variable declarations and comments have no Instr Exec counts.

The following figure shows an example of the Source view opened.



For source code lines that do not have any coverage information associated with them, the rows are blank and have no numbers in any of the fields. These empty rows can occur because of the following reasons:

- Comments, blank lines, declarations, and other language constructs do not contain executable code.
- Compiler optimizations have deleted the code corresponding to the lines due to either of the following reasons:
  - The code will never be executed (dead code).
  - The code can be executed but is redundant.

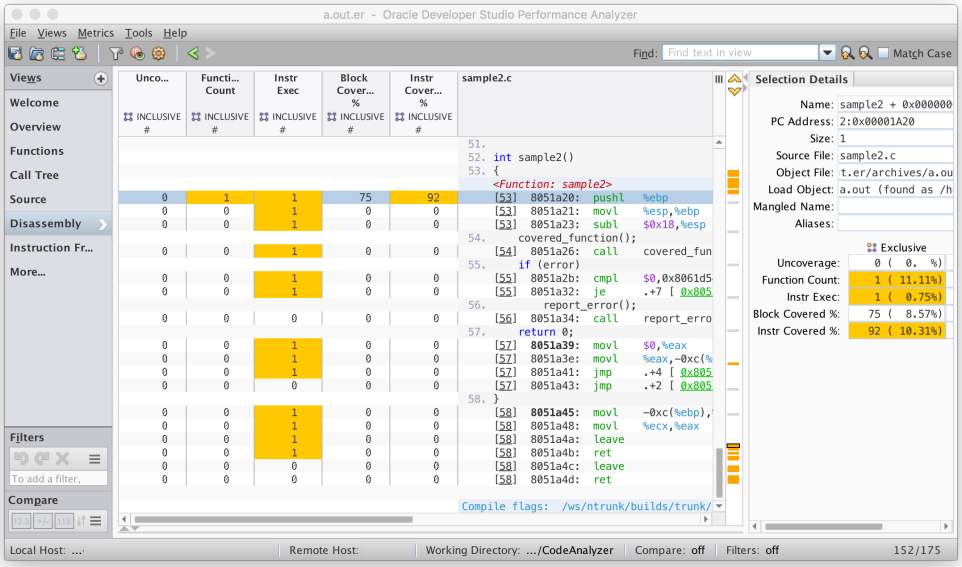
For more information, see “Limitations When Using uncover” on page 80.

Disassembly View

If you select a line in the Source view and then select the Disassembly view, Performance Analyzer tries to find the selected line in the binary and display its disassembly.

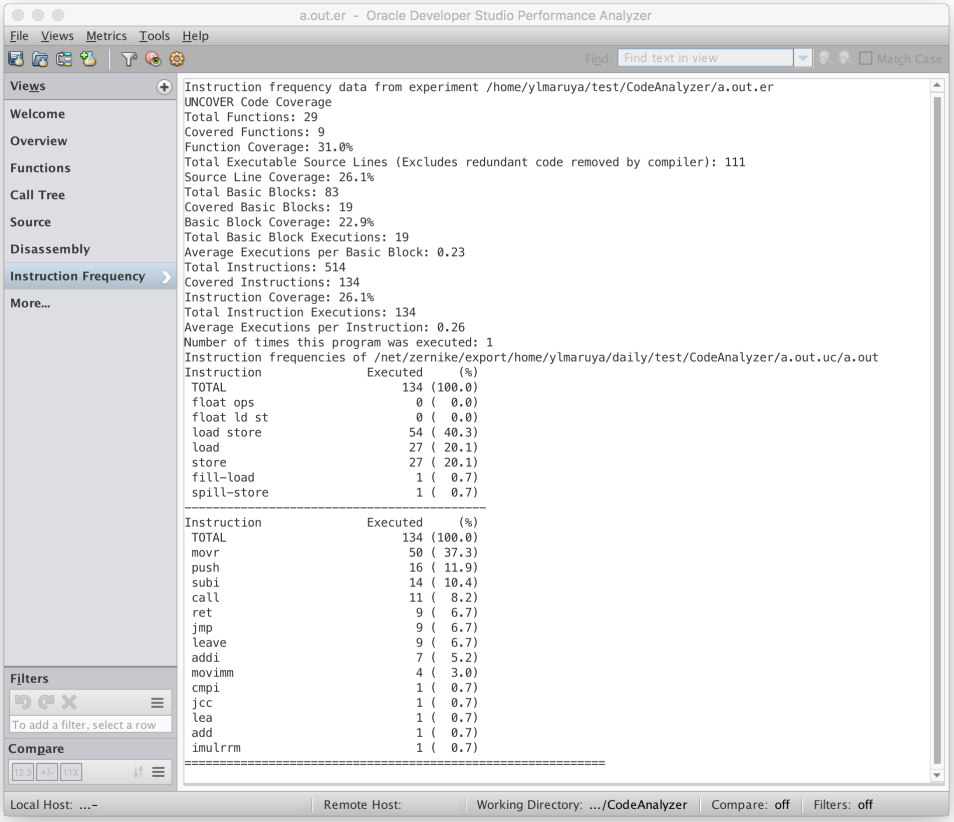
Tip - If you do not see Disassembly in the Views pane, then select More Views... and check the Disassembly option.

The Instr Exec counter in this view shows the number of times each instruction was executed:



Instruction Frequency View

The Instruction Frequency view displays the overall coverage summary:



Understanding the ASCII Coverage Report

If you specify the `-t` option when you generate the coverage report from the coverage data directory, `uncover` writes a coverage report to the specified ASCII (text file).

EXAMPLE 4 Sample ASCII Coverage Report

The following example shows a sample ASCII coverage report:

```
UNCOVER Code Coverage
Total Functions: 95
```

```
Covered Functions: 58
Function Coverage: 61.1%
Total Basic Blocks: 568
Covered Basic Blocks: 258
Basic Block Coverage: 45.4%
Total Basic Block Executions: 564,812,760
Average Executions per Basic Block: 994,388.66
Total Instructions: 6,201
Covered Instructions: 3,006
Instruction Coverage: 48.5%
Total Instruction Executions: 4,760,934,518
Average Executions per Instruction: 767,768.83
Number of times this program was executed: unavailable
Functions sorted by metric: Exclusive Uncoverage

Excl.       Excl.     Excl.      Excl.       Name
Uncoverage  Function  Block      Instr
Count     Covered %  Covered %
13404       6004876   5464       5384        <Total>
1036             0      0          0        main
980              0      0          0        iofile
748              0      0          0        do_vforkexec
732              0      0          0        callso
708              0      0          0        do_forkexec
648              0      0          0        callsx
644              0      0          0        sigprof
644              0      0          0        sigprofh
556              0      0          0        do_chdir
548              0      0          0        correlate
492              0      0          0        do_popen
404              0      0          0        pagethrash
384              0      0          0        so_cputime
384              0      0          0        sx_cputime
348              0      0          0        itimer_realprof
336              0      0          0        ldso
304              0      0          0        hrv
300              0      0          0        do_system
300              0      0          0        do_burncpu
300              0      0          0        sx_burncpu
288              0      0          0        forkcopy
276              0      0          0        masksignals
256              0      0          0        sigprof_handler
256              0      0          0        sigprof_sigaction
216              0      0          0        do_exec
196              0      0          0        iotest
176              0      0          0        closeso
156              0      0          0        gethrustime
144              0      0          0        forkchild
```

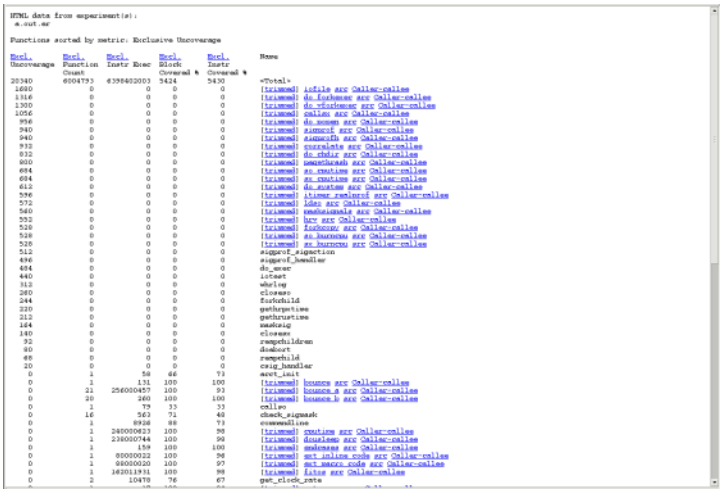
```
144             0      0          0        gethrpxtime
136             0      0          0        whrlog
112             0      0          0        masksig
92              0      0          0       closesx
84              0      0          0       reapchildren
36              0      0          0       reapchild
32              0      0          0       doabort
8               0      0          0      csig_handler
0               1     66         72      acct_init
0               1    100        100      bounce
0              63    100         96      bounce_a
0              60    100        100      bounce-b
0              16     71         58      check_sigmask
0               1     83         77      commandline
0               1    100         98      cputime
0               1    100         98      dousleep
0               1    100        100      endcases
0               1    100         95      ext_inline_code
0               1    100         96      ext_macro_code
0               1    100         99      fitos
0               2     81         80      get_clock_rate
0               1    100        100      get_ncpus
0               1    100        100      gpf
0               1    100        100      gpf_a
0               1    100        100      gpf_b
0              10    100         93      gpf_work
0               1    100         97      icputime
0               1    100         96      inc_body
0               1    100         96      inc_brace
0               1    100         95      inc_entry
0               1    100         95      inc_exit
0               1    100         96      inc_func
0               1    100         94      inc_middle
0               1     57         72      init_micro_acct
0               1     50         43      initcksig
0               1    100         95      inline_code
0               1    100         95      macro_code
0               1    100         98      muldiv
0         6000000    100        100      my_irand
0               1    100         98      naptime
0              19     50         83      prdelta
0              21    100        100      prhrdelta
0              21    100        100      prhrvdelta
0               1    100        100      prtime
0             552    100         98      real_recurse
0               1    100        100      recurse
0               1    100        100      recursedeep
0               1    100         95      s_inline_code
```

```
0             1    100        100        sigtime
0             1    100         95        sigtime_handler
0            19    100        100        snaptod
0             1    100        100        so_init
0             2     66         75        stpwtch_alloc
0             1    100        100        stpwtch_calibrate
0             2     75         66        stpwtch_print
0          2002    100        100        stpwtch_start
0          2000     90         91        stpwtch_stop
0             1    100        100        sx_init
0             1    100         99        systime
0             3    100         95        tailcall_a
0             3    100         95        tailcall_b
0             3    100         95        tailcall_c
0             1    100        100        tailcallopt
0             1    100         97        underflow
0            21     75         71        whrvlog
0            19    100        100        wlog

Instruction frequency data from experiment a.out.er

Instruction frequencies of /export/home1/synprog/a.out.uc
Instruction               Executed     ()
TOTAL                  4760934518 (100.0)
float ops              2383657378 ( 50.1)
float ld st            1149983523 ( 24.2)
load store             1542440573 ( 32.4)
load                    882693735 ( 18.5)
store                   659746838 ( 13.9)
-----------------------------------------
Instruction               Executed     ()      Annulled   In Delay Slot
TOTAL                  4760934518 (100.0)
add                     713013787 ( 15.0)           16         1501335
subcc                   558774858 ( 11.7)            0            6002
br                      558769261 ( 11.7)            0               0
stf                     432500661 (  9.1)          726        36299281
ldf                     408226488 (  8.6)           40       103000396
faddd                   391230847 (  8.2)            0               0
fdtos                   366200726 (  7.7)            0               0
fstod                   360200000 (  7.6)            0               0
lddf                    288250336 (  6.1)          500       282200229
stw                     138028738 (  2.9)        26002        25974065
lduw                    118004305 (  2.5)           71        94000270
ldx                      68212446 (  1.4)            0            2000
stx                      68211370 (  1.4)            7        23532716
fitod                    36026002 (  0.8)            0               0
sethi                    36002986 (  0.8)            0             228
fdtoi                    30000001 (  0.6)            0               0
```

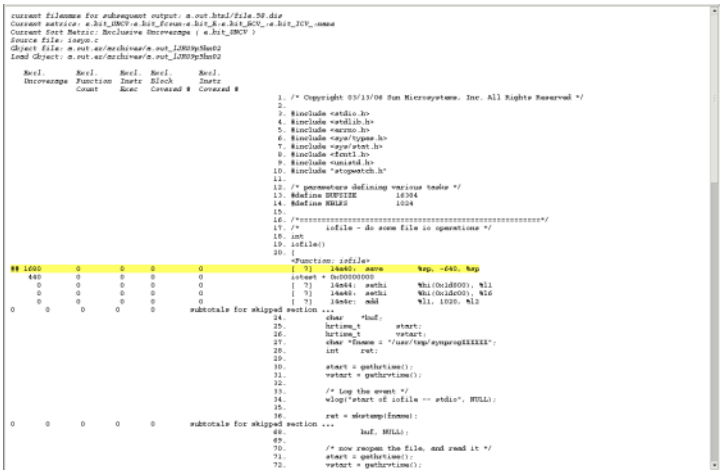
```
fdivd                   26000088 (  0.5)           0              0
call                    22250348 (  0.5)           0              0
srl                     21505246 (  0 5)           0             21
stdf                    21006038 (  0.4)           0              0
or                      19464766 (  0.4)           0       10981277
fmuls                    6004907 (  0.3)           0              0
jmpl                     6004853 (  0.1)           0              0
save                     6004852 (  0.1)           0              0
restore                  6002294 (  0.1)           0        6004852
sub                      6000019 (  0.1)           0              0
xor                      6000000 (  0.1)           0              0
fitos                    6000000 (  0.1)           0              0
fstoi                    6000000 (  0.1)           0              0
and                      6000000 (  0.1)           0              0
andn                     6000000 (  0.1)           0              0
sll                      3505225 (  0.1)           0              0
nop                      3505219 (  0.1)           0        3505219
fxtod                       7763 (  0.0)           0              0
bpr                         6000 (  0.0)           0              0
fcmped                      4837 (  0.0)           0              0
fbr                         4837 (  0.0)           0              0
fmuld                       2850 (  0.0)           0              0
orcc                         383 (  0.0)           0              0
sra                          241 (  0.0)           0              0
ldsb                         160 (  0.0)           0              0
mulx                          87 (  0.0)           0              0
stb                           31 (  0.0)           0              0
mov                           21 (  0.0)           0              0
fdtox                         15 (  0.0)           0              0
=============================================================
```

Understanding the HTML Coverage Report

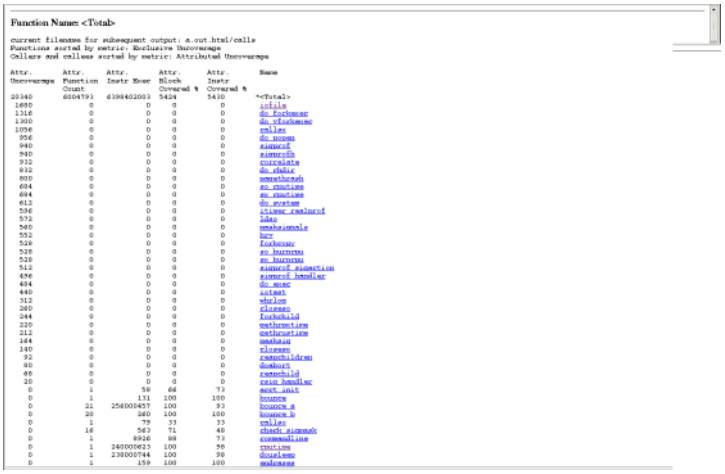
The HTML report is similar to the report displayed in Performance Analyzer:



Click the function name link or the `trimmed` link for a function to display, the disassembly data for that function:



Click the `Caller-callee` link for a function to display the Caller-Callee data:



Limitations When Using `uncover`

This section describes known limitations when using `uncover`.

Only Annotated Code Can Be Instrumented

The `uncover` utility can only instrument code that has been described in "Requirements for Using `uncover`" on page 63. Non-annotated code might come from assembly language code linked into the binary or from modules compiled with older compilers or operating systems than those listed in that section.

`uncover` cannot instrument assembly language modules or functions that contain `asm` statements or `.il` templates.

Compiler Options Affect Generated Code

`uncover` is incompatible with binaries built with any of the following compiler options:

- `-p`
- `-pg`
- `-qp`

- -xpg
- -xlinkopt

Machine Instructions Might Differ From Source Code

The uncover utility operates on machine code. It finds coverage of machine instructions and then correlates this coverage with source code. Some source code statements do not have associated machine instructions, so uncover might appear to not report coverage for such statements.

EXAMPLE 5 Simple Example

Consider the following code fragment:

```
#define A 100
#define B 200
...
if (A>B) {
...
}
```

You might expect uncover to report a non-zero execution count for the if statement. However, the compiler is likely to remove this code. uncover will not detect it during instrumentation and no coverage will be reported for these instructions.

EXAMPLE 6 Dead Code Example

The following example shows dead code:

```
1  void foo()
2  {
3     A();
4     return;
5     B();
6     C();
7     D();
8     return;
9   }
```

Corresponding assembly shows that calls to B,C,D are deleted because this code is never executed.

```
foo:
.L900000109:
/* 000000          2 */         save    %sp,-96,%sp
/* 0x0004          3 */         call    A       ! params =      ! Result =
/* 0x0008            */         nop
/* 0x000c          8 */         ret     ! Result =
/* 0x0010            */         restore %g0,%g0,%g0
```

Therefore, no coverage will be reported for lines 5 through 7.

```
Excl.       Excl.     Excl.  Excl.      Excl.
Uncoverage  Function  Instr  Block      Instr
            Count     Exec   Covered %  Covered %
1.  void foo()
## 0           1         1      100        100
2.   {
<Function: foo
## 0           0         2        0          0
3.      A();
4.      return;
5.      B();
6.      C();
7.      D();
8.     return;
## 0           0         2        0          0
9.    }
```

EXAMPLE 7 Redundant Code Example

The following example shows redundant code:

```
1  int g;
2  int foo() {
3    int x;
4    x = g;
5    for (int i=0; i<100; i++)
6        x++;
7    return x;
8  }
```

At low optimization levels, the compiler can generate code for all the lines:

```
foo:
.L900000107:
/* 000000          3 */         save    %sp,-112,%sp
/* 0x0004          5 */         sethi   %hi(g),%l1
/* 0x0008            */         ld      [%l1+%lo(g)],%l3 ! volatile
/* 0x000c            */         add     %l1,%lo(g),%l2
```

```
/* 0x0010          6 */         st      %g0,[%fp-12]
/* 0x0014          5 */         st      %l3,[%fp-8]
/* 0x0018          6 */         ld      [%fp-12],%l4
/* 0x001c            */         cmp     %l4,100
/* 0x0020            */         bge,a,pn        %icc,.L900000105
/* 0x0024          8 */         ld      [%fp-8],%l1
.L17:
/* 0x0028          7 */         ld      [%fp-8],%l1
.L900000104:
/* 0x002c          6 */         ld      [%fp-12],%l3
/* 0x0030          7 */         add     %l1,1,%l2
/* 0x0034            */         st      %l2,[%fp-8]
/* 0x0038          6 */         add     %l3,1,%l4
/* 0x003c            */         st      %l4,[%fp-12]
/* 0x0040            */         ld      [%fp-12],%l5
/* 0x0044            */         cmp     %l5,100
/* 0x0048            */         bl,a,pn %icc,.L900000104
/* 0x004c          7 */         ld      [%fp-8],%l1
/* 0x0050          8 */         ld      [%fp-8],%l1
.L900000105:
/* 0x0054          8 */         st      %l1,[%fp-4]
/* 0x0058            */         ld      [%fp-4],%i0
/* 0x005c            */         ret     ! Result =  %i0
/* 0x0060            */         restore %g0,%g0,%g0
```

At high optimization levels, most of the executable source lines do not have any corresponding instructions:

```
foo:
/* 000000          5 */         sethi   %hi(g),%o5
/* 0x0004            */         ld      [%o5+%lo(g)],%o4
/* 0x0008          8 */         retl    ! Result =  %o0
/* 0x000c          5 */         add     %o4,100,%o0
```

Therefore, no coverage will be reported for some lines.

```
Excl.       Excl.     Excl.  Excl.      Excl.
Uncoverage  Function  Instr  Block      Instr
            Count     Exec   Covered %  Covered %
1. int g;
0           0         0        0          0
2. int foo() {
<Function foo>
3.   int x;
4.   x = g;

Source loop below has tag L1
Induction variable substitution performed on L1
L1 deleted as dead code
```

```
## 0           1         3      100        100
5.   for (int i=0; i<100; i++)
6.       x++;
7.   return x;
0           0         1        0          0
8. }
```

Index

I

L