Oracle® Solaris Studio 12.4: Code Analyzer Tutorial

October 2014

Part No: E37091

This tutorial uses a sample program to demonstrate how to use the Oracle Solaris Studio compilers, the `discover` memory error discovery tool, the `uncover` code coverage tool, and the Code Analyzer GUI to find and correct common programming errors, dynamic memory access errors, and code coverage issues.

Introduction to Code Analyzer

Oracle Solaris Studio Code Analyzer is an integrated set of tools designed to help developers of C and C++ applications for Oracle Solaris produce secure, robust, and quality software. It works in conjunction with the Oracle Solaris Studio compilers; `discover`, a memory error discovery tool; and `uncover`, a code coverage tool.

Code Analyzer includes three types of analysis:

- Static code checking as part of compilation
- Dynamic memory access checking
- Code coverage analysis

Static code checking detects common programming errors in your code during compilation. A new compiler option leverages the Oracle Solaris Studio compilers' control and data flow analysis frameworks to analyze an application for potential programming and security flaws.

The Code Analyzer uses dynamic memory data collected by `discover` to find memory-related errors when you run your application. It uses data collected by `uncover` to measure code coverage.

In addition providing access to each individual type of analysis, Code Analyzer integrates static code checking with dynamic memory access analysis and code coverage analysis, to enable you to find many important errors in your applications that cannot be found by other error detection tools working separately.

Getting the Sample Application

The source code for the sample program is available in the sample applications zip file on the Oracle Solaris Studio 12.4 downloads web page at `http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html`. After accepting the license and downloading, you can unpack it in a directory of your choice.

The `sample` application is located in the `CodeAnalyzer` subdirectory of the `SolarisStudioSampleApplications` directory.

The `sample` directory contains the following source code files:

```
main.c
previse_1.c
previse_all.c
```

```
sample1.c
sample2.c
sample3.c
sample4.c
```

Collecting and Displaying Data

You can use the Code Analyzer tools to collect up to three types of data.

Collecting and Displaying Static Error Data

When you build a binary using the `-xprevise` compiler option, the compiler automatically extracts static errors and puts the data in a `static` subdirectory in a *binary-name*`.analyze` directory in the same directory as the source code. For a list of the types of static errors found by the compiler, see "Static Code Issues" on page 16.

1. In your `sample` directory, build the application by typing the following:

Note - The `-xprevise` option is synonymous with `-xanalyze=code`, which is being deprecated.

- On Oracle Solaris:

```
$ cc -xprevise main.c previse*.c sample1.c sample2.c sample3.c
```

- On Oracle Linux:

```
$ cc -xannotate -xprevise main.c previse*.c sample1.c sample2.c sample3.c
```

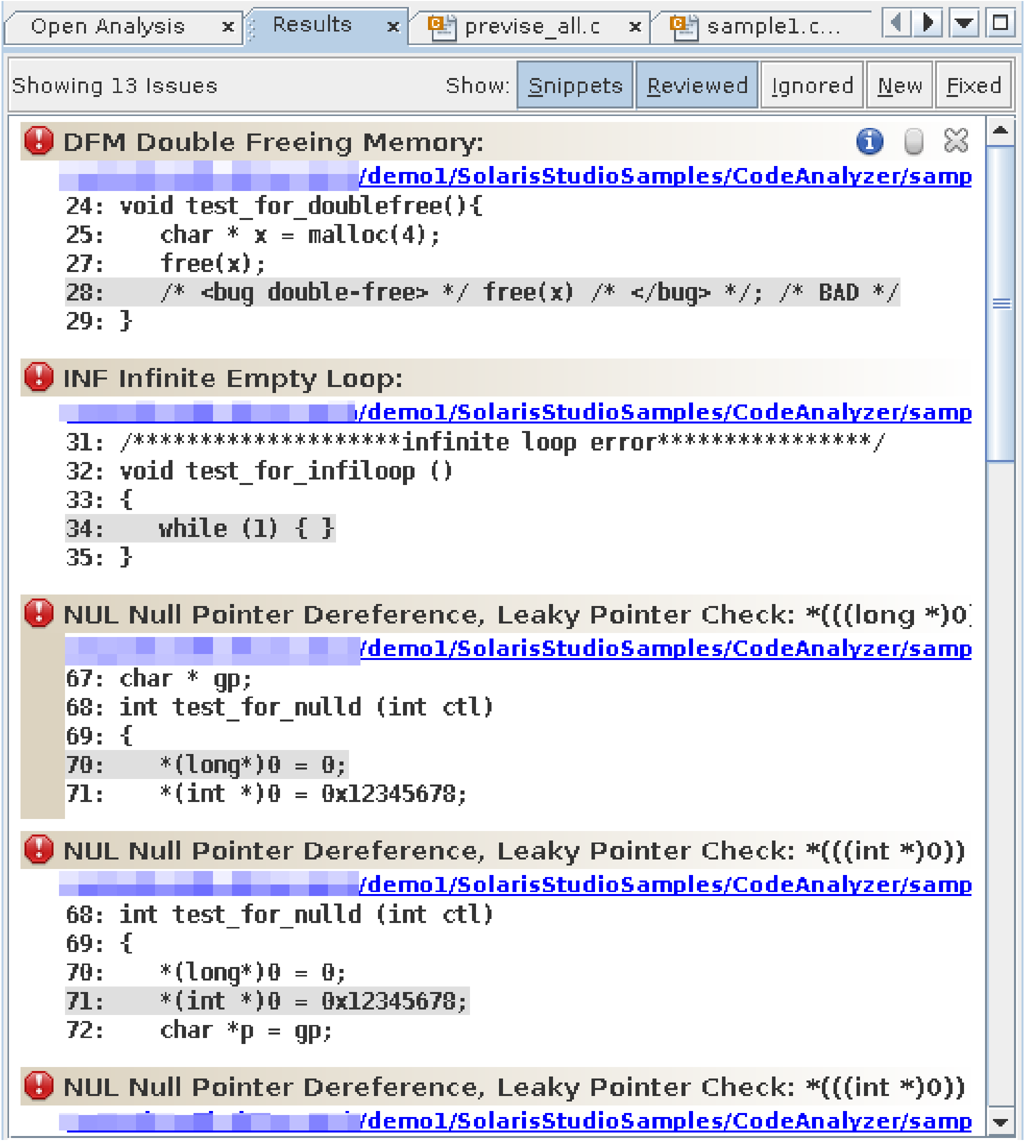
Note - You are not compiling `sample4.c` for this portion of the tutorial.

The static error data is written to the `sample/a.out.analyze/static` directory.

2. Start the Code Analyzer GUI to view the results:

```
$ code-analyzer a.out &
```

3. The Code Analyzer GUI opens and the Results tab displays the static code issues found during compilation. The text at the top left of the Results tab indicates that thirteen static code issues were found.



Showing 13 Issues

DFM Double Freeing Memory:

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/samp

```
void test_for_doublefree(){
    char * x = malloc(4);
    free(x);
    /* <bug double-free> */ free(x) /* </bug> */; /* BAD */
}
```

INF Infinite Empty Loop:

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/samp

```
/*******************infinite loop error****************/
void test_for_infiloop ()
{
    while (1) { }
}
```

NUL Null Pointer Dereference, Leaky Pointer Check: *(((long *)0)

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/samp

```
char * gp;
int test_for_nulld (int ctl)
{
    *(long*)0 = 0;
    *(int *)0 = 0x12345678;
```

NUL Null Pointer Dereference, Leaky Pointer Check: *(((int *)0))

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/samp

```
int test_for_nulld (int ctl)
{
    *(long*)0 = 0;
    *(int *)0 = 0x12345678;
    char *p = gp;
```

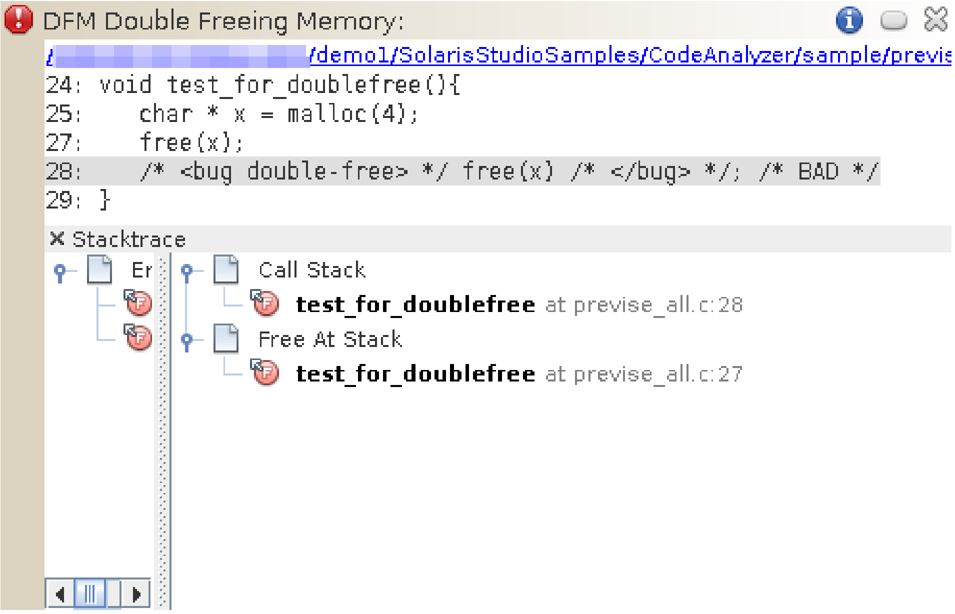
NUL Null Pointer Dereference, Leaky Pointer Check: *(((int *)0))

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/samp

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

4. To see more information about the first issue, a Double Freeing Memory error, click the error icon ❗.

   The stack trace for the issue opens displaying the error path:



DFM Double Freeing Memory:

/[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/previs

```
24: void test_for_doublefree(){
25:     char * x = malloc(4);
27:     free(x);
28:     /* <bug double-free> */ free(x) /* </bug> */; /* BAD */
29: }
```

✕ Stacktrace
Er
Call Stack
test_for_doublefree at previse_all.c:28
Free At Stack
test_for_doublefree at previse_all.c:27

Notice that when you opened the stack trace, the icon in the upper right corner of the issue changed from ▯ to ▭ to indicate that you have reviewed the issue.

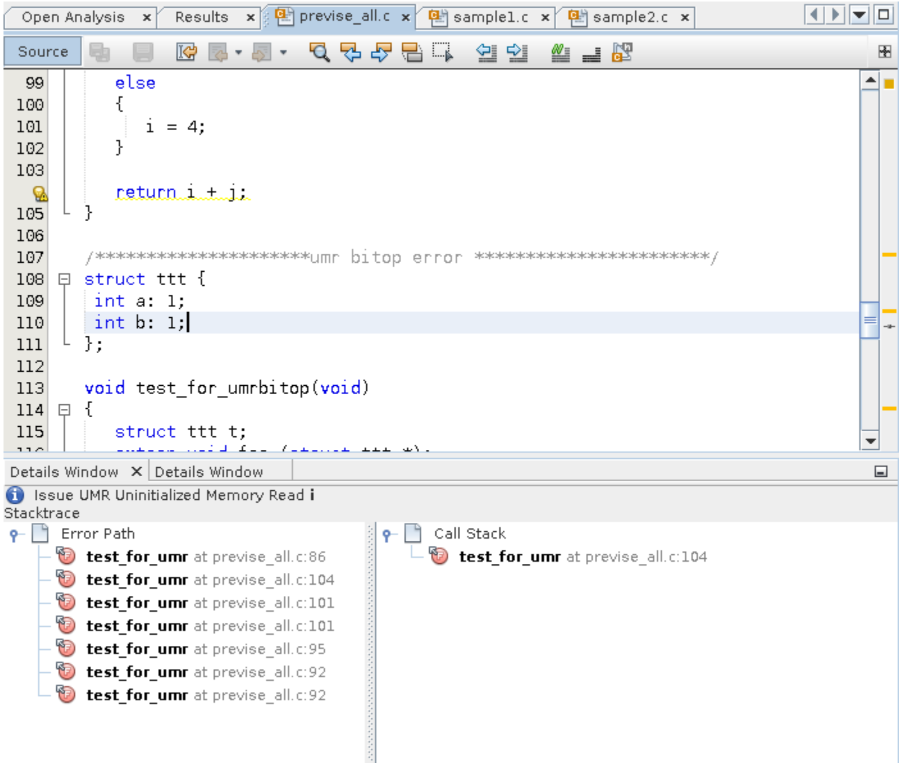
Note - You can hide the issues you have reviewed by clicking the Reviewed button [Reviewed] at the top of the Results tab. Clicking the button again unhides the issues.

---

5. Click the same error icon to close the stack trace.
6. Click the warning icon ⚠ of one of the Uninitialized Memory Read warnings to open the stack trace.

   The error path for this issue contains many more function calls than the one for the Double Freeing Memory issue.
7. Double click the first function call.

   The source file opens with that call highlighted. The error path is displayed in a Details Window below the source code.



```
        else
        {
            i = 4;
        }

        return i + j;
}

/**********************umr bitop error ***********************/
struct ttt {
 int a: 1;
 int b: 1;
};

void test_for_umrbitop(void)
{
    struct ttt t;
```

Issue UMR Uninitialized Memory Read

Stacktrace

Error Path

- test_for_umr at previse_all.c:86
- test_for_umr at previse_all.c:104
- test_for_umr at previse_all.c:101
- test_for_umr at previse_all.c:101
- test_for_umr at previse_all.c:95
- test_for_umr at previse_all.c:92
- test_for_umr at previse_all.c:92

Call Stack

- test_for_umr at previse_all.c:104

8. Double-click the other function calls in the error path to follow the path through the code that leads to the error.

9. Click the Info button ⓘ to the left of the issue description to see more information about the UMR error type.

   A description of the error type, including a code example and possible causes, is displayed in the online help browser.

10. Close the Code Analyzer GUI by pressing the X in the upper right corner.

Collecting and Displaying Dynamic Memory Usage Data

Regardless of whether you have collected static data, you can compile, instrument, and run your application to collect dynamic memory access data. For a list of the dynamic memory access errors found by instrumenting your application with `discover` and then running it, see "Dynamic Memory Access Issues" on page 16.

1. In your `sample` directory, build the sample application with the `-g` option.

   This option generates debug information that enables Code Analyzer to display source code and line number information for errors and warnings.

   - On Oracle Solaris:

     ```
     $ cc -g main.c previse*.c sample1.c sample2.c sample3.c
     ```

   - On Oracle Linux:

     ```
     $ cc -xannotate -g main.c previse*.c sample1.c sample2.c sample3.c
     ```

Note - You are not compiling `sample4.c` for this portion of the tutorial.

---

2. Save a copy of the binary to use when you collect coverage data because you cannot instrument a binary that is already instrumented.

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

3. Instrument the binary with `discover`.

   ```
   $ discover -a a.out
   ```

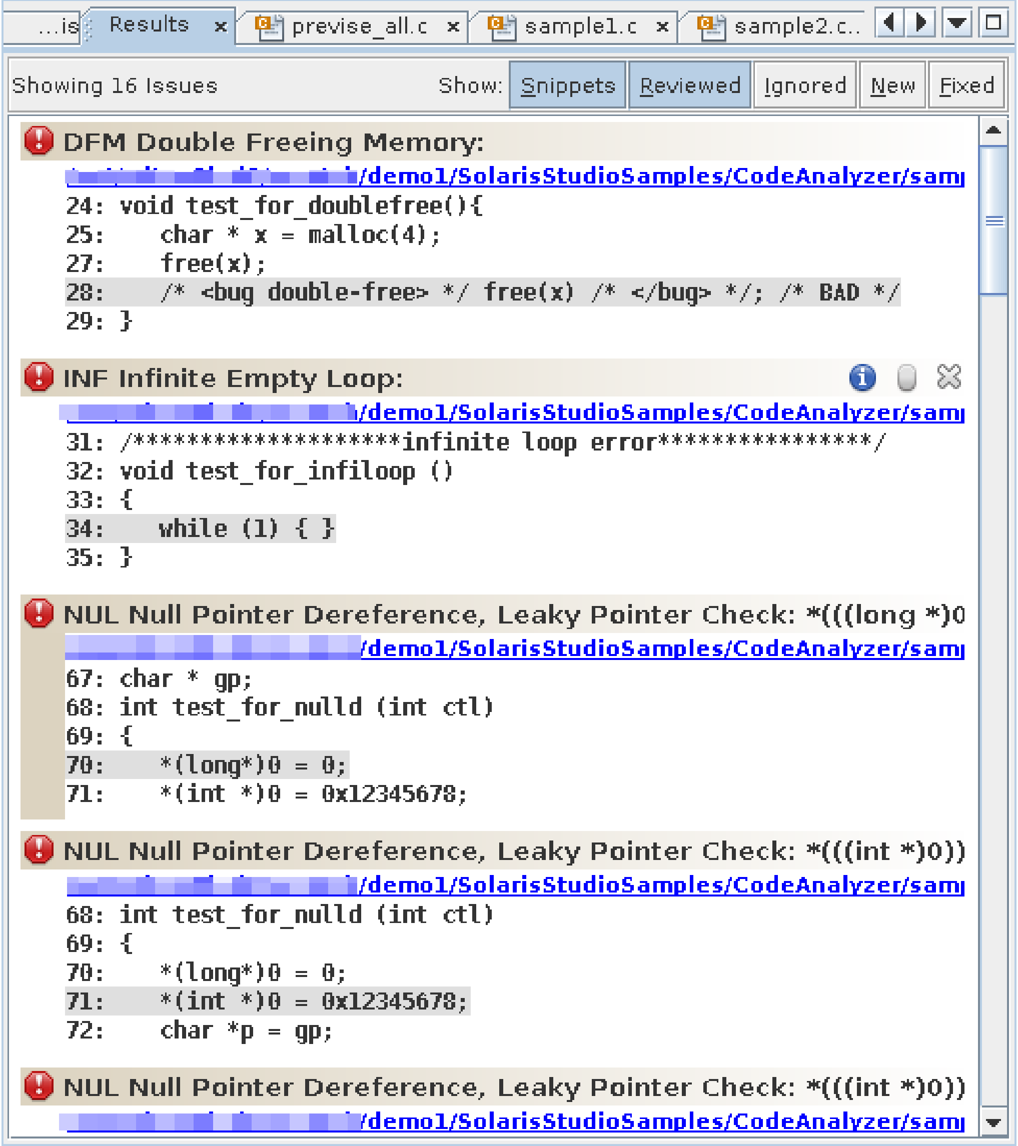
4. Run the instrumented binary to collect the dynamic memory access data.

   ```
   $ ./a.out
   ```

   The dynamic memory access error data is written to the `sample/a.out.analyze/dynamic` directory.

5. Start the Code Analyzer GUI to view the results.

   ```
   $ code-analyzer a.out &
   ```



Showing 16 Issues

DFM Double Freeing Memory:

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sam

```
void test_for_doublefree(){
    char * x = malloc(4);
    free(x);
    /* <bug double-free> */ free(x) /* </bug> */; /* BAD */
}
```

INF Infinite Empty Loop:

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sam

```
/********************infinite loop error****************/
void test_for_infiloop ()
{
    while (1) { }
}
```

NUL Null Pointer Dereference, Leaky Pointer Check: *(((long *)0

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sam

```
char * gp;
int test_for_nulld (int ctl)
{
    *(long*)0 = 0;
    *(int *)0 = 0x12345678;
```

NUL Null Pointer Dereference, Leaky Pointer Check: *(((int *)0))

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sam

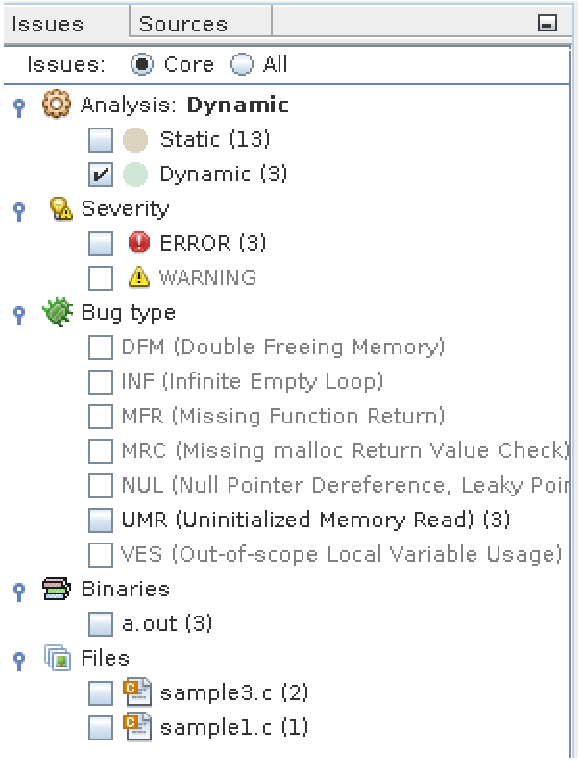
```
int test_for_nulld (int ctl)
{
    *(long*)0 = 0;
    *(int *)0 = 0x12345678;
    char *p = gp;
```

NUL Null Pointer Dereference, Leaky Pointer Check: *(((int *)0))

[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sam

The Results tab shows both static issues and dynamic memory issues. The background color behind an issue description indicates whether it is a static code issue (tan) or a dynamic memory access issue (pale green).

6. To filter the results and show just the dynamic memory issues, select the Dynamic option in the Issues tab.



Issues | Sources

Issues: ● Core ○ All

Analysis: **Dynamic**

- □ Static (13)
- ☑ Dynamic (3)

Severity

- □ ERROR (3)
- □ WARNING

Bug type

- □ DFM (Double Freeing Memory)
- □ INF (Infinite Empty Loop)
- □ MFR (Missing Function Return)
- □ MRC (Missing malloc Return Value Check)
- □ NUL (Null Pointer Dereference, Leaky Poi
- □ UMR (Uninitialized Memory Read) (3)
- □ VES (Out-of-scope Local Variable Usage)

Binaries

- □ a.out (3)

Files

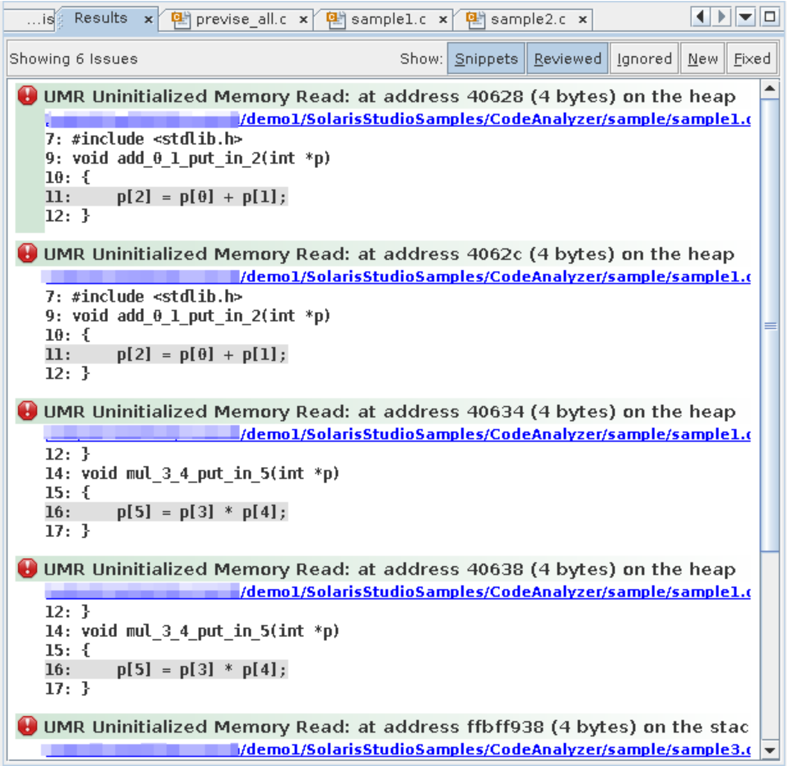
- □ sample3.c (2)
- □ sample1.c (1)

The Results tab now shows just the three core dynamic memory issues.

Note - Core issues are issues that, when fixed, are likely to eliminate the other issues. A core issue usually combines several of the issues listed in the All view because, for example, those issues have a common allocation point or occur at the same data address in the same function.

---

7. To see all of the dynamic memory issues, select the All radio button at the top of the Issues tab. The Results tab now displays six dynamic memory issues.



```
UMR Uninitialized Memory Read: at address 40628 (4 bytes) on the heap
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c
7: #include <stdlib.h>
9: void add_0_1_put_in_2(int *p)
10: {
11:     p[2] = p[0] + p[1];
12: }
UMR Uninitialized Memory Read: at address 4062c (4 bytes) on the heap
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c
7: #include <stdlib.h>
9: void add_0_1_put_in_2(int *p)
10: {
11:     p[2] = p[0] + p[1];
12: }
UMR Uninitialized Memory Read: at address 40634 (4 bytes) on the heap
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c
12: }
14: void mul_3_4_put_in_5(int *p)
15: {
16:     p[5] = p[3] * p[4];
17: }
UMR Uninitialized Memory Read: at address 40638 (4 bytes) on the heap
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c
12: }
14: void mul_3_4_put_in_5(int *p)
15: {
16:     p[5] = p[3] * p[4];
17: }
UMR Uninitialized Memory Read: at address ffbff938 (4 bytes) on the stac
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample3.c
```

Look at the three issues that were added to the display and see how they are related to the core issues. Fixing the cause of the first issue in the display is likely also to eliminate the second and third issues.

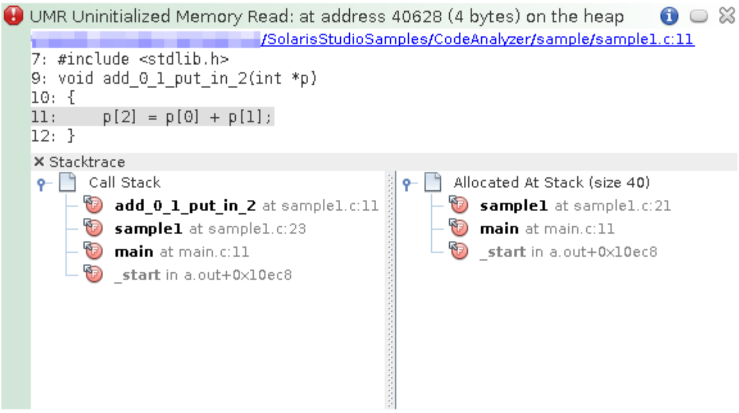
To hide the other dynamic memory access issues while you investigate the first one, click the Ignore button ☒for each of the issues.

Note - You can later redisplay the closed issues by clicking the Ignored button at the top of the Results tab.

---

8. Investigate the first issue by clicking the error icon to display the stack trace.

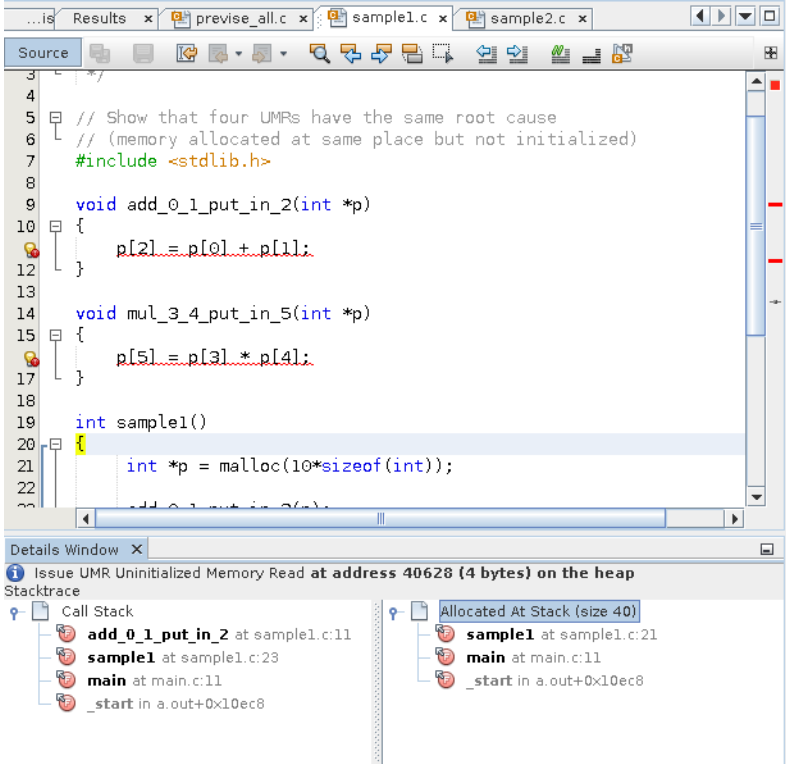
   For this issue, the stack trace includes the Call Stack and the Allocated At Stack.



```
UMR Uninitialized Memory Read: at address 40628 (4 bytes) on the heap
[redacted]/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c:11
7: #include <stdlib.h>
9: void add_0_1_put_in_2(int *p)
10: {
11:     p[2] = p[0] + p[1];
12: }
× Stacktrace
Call Stack
    add_0_1_put_in_2 at sample1.c:11
    sample1 at sample1.c:23
    main at main.c:11
    _start in a.out+0x10ec8
Allocated At Stack (size 40)
    sample1 at sample1.c:21
    main at main.c:11
    _start in a.out+0x10ec8
```

9. Double-click function calls in the stacks to see the associated lines in the source file.

   When the source file opens, the stack trace is displayed in a Details window below the file.



```
 */

// Show that four UMRs have the same root cause
// (memory allocated at same place but not initialized)
#include <stdlib.h>

void add_0_1_put_in_2(int *p)
{
    p[2] = p[0] + p[1];
}

void mul_3_4_put_in_5(int *p)
{
    p[5] = p[3] * p[4];
}

int sample1()
{
    int *p = malloc(10*sizeof(int));

    add_0_1_put_in_2(p);
```

Issue UMR Uninitialized Memory Read **at address 40628 (4 bytes) on the heap**

Stacktrace

Call Stack
- **add_0_1_put_in_2** at sample1.c:11
- **sample1** at sample1.c:23
- **main** at main.c:11
- **_start** in a.out+0x10ec8

Allocated At Stack (size 40)
- **sample1** at sample1.c:21
- **main** at main.c:11
- **_start** in a.out+0x10ec8

10. Close the Code Analyzer GUI by pressing the X in the upper right corner.

Collecting and Displaying Code Coverage Data

Regardless of whether you have collected static data or dynamic memory access data, you can compile, instrument, and run your application to collect code coverage data. Note that when you built application with the -g option before you collected dynamic memory error data, you saved a copy of the binary before instrumenting it.

1. Copy the saved binary to instrument for coverage data collection.

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

2. Instrument the binary with uncover.

```
$ uncover a.out
```

3. Run the instrumented binary to collect the code coverage data.

```
$ ./a.out
```

   The code coverage data is written to an `a.out.uc` directory in your `sample` directory.

4. Run `uncover` on the `a.out.uc` directory.

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

   The code coverage data is written to the `sample/a.out.analyze/uncover` directory..

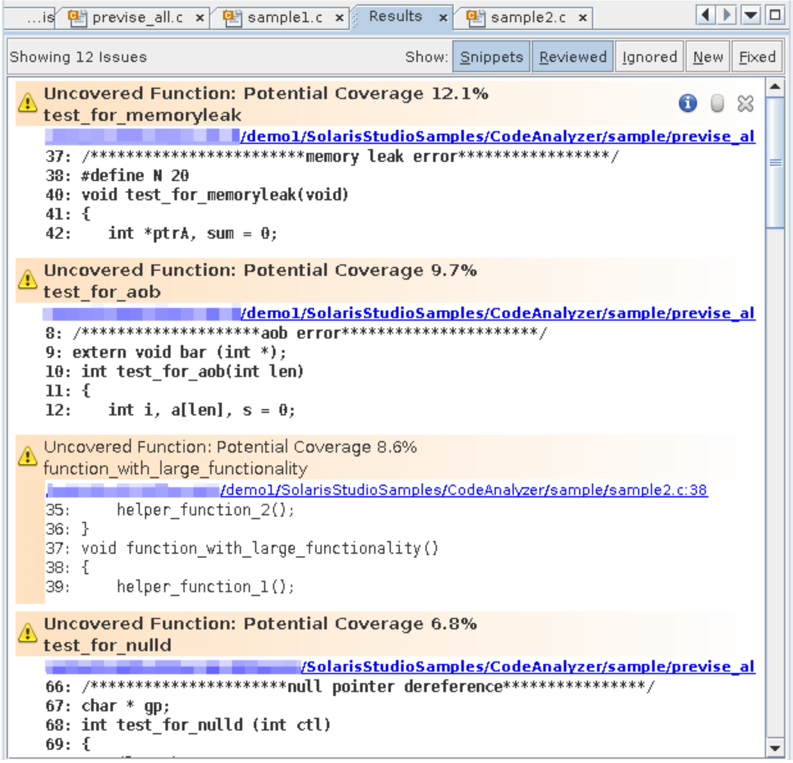
5. Start the Code Analyzer GUI to view the results:

```
$ code-analyzer a.out &
```

   The Results tab shows static issues, dynamic memory issues, and code coverage issues.

6. To filter the results and show just the code coverage issues, select the Coverage option in the Issues tab.

   The Results tab now shows just the twelve code coverage issues. The description of each issue includes a potential coverage percentage, which indicates the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.



Uncovered Function: Potential Coverage 12.1%
test_for_memoryleak
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/previse_al

```
/************************memory leak error*****************/
#define N 20
void test_for_memoryleak(void)
{
    int *ptrA, sum = 0;
```

Uncovered Function: Potential Coverage 9.7%
test_for_aob
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/previse_al

```
/*******************aob error**********************/
extern void bar (int *);
int test_for_aob(int len)
{
    int i, a[len], s = 0;
```

Uncovered Function: Potential Coverage 8.6%
function_with_large_functionality
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample2.c:38

```
    helper_function_2();
}
void function_with_large_functionality()
{
    helper_function_1();
```

Uncovered Function: Potential Coverage 6.8%
test_for_nulld
[redacted]/SolarisStudioSamples/CodeAnalyzer/sample/previse_al

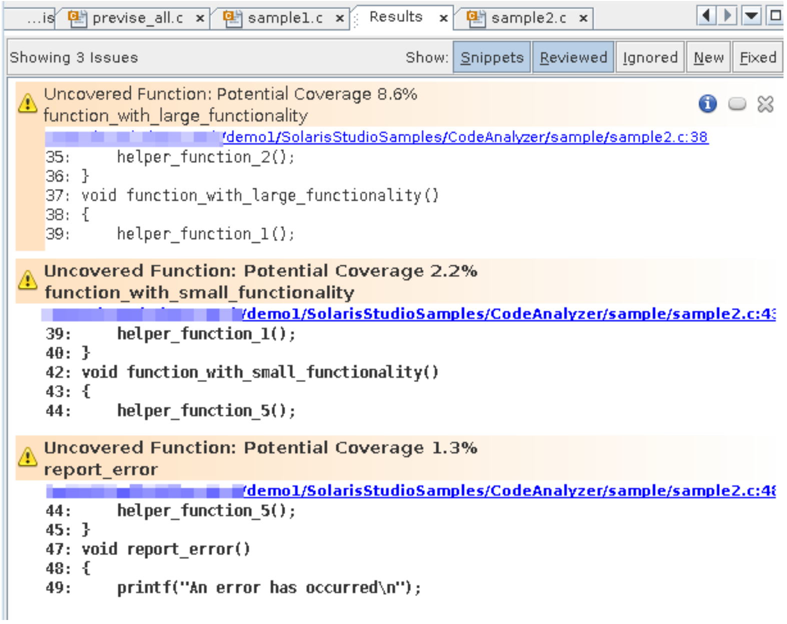
```
/**********************null pointer dereference****************/
char * gp;
int test_for_nulld (int ctl)
{
```

Tip - To see all the issues without scrolling up and down, click the Snippets button Snippets at the top of the Results tab to hide the code snippets.

   In the Issues tab, nine of the coverage issues are in the `previse_all.c` source file, three are in `sample2.c`, and one is in `previse_1.c`.

7. To show just the issues for the `sample2.c` file, select the option for that file on the Issues tab.

   The Results tab now shows just the three code coverage issues found in `sample2.c`.

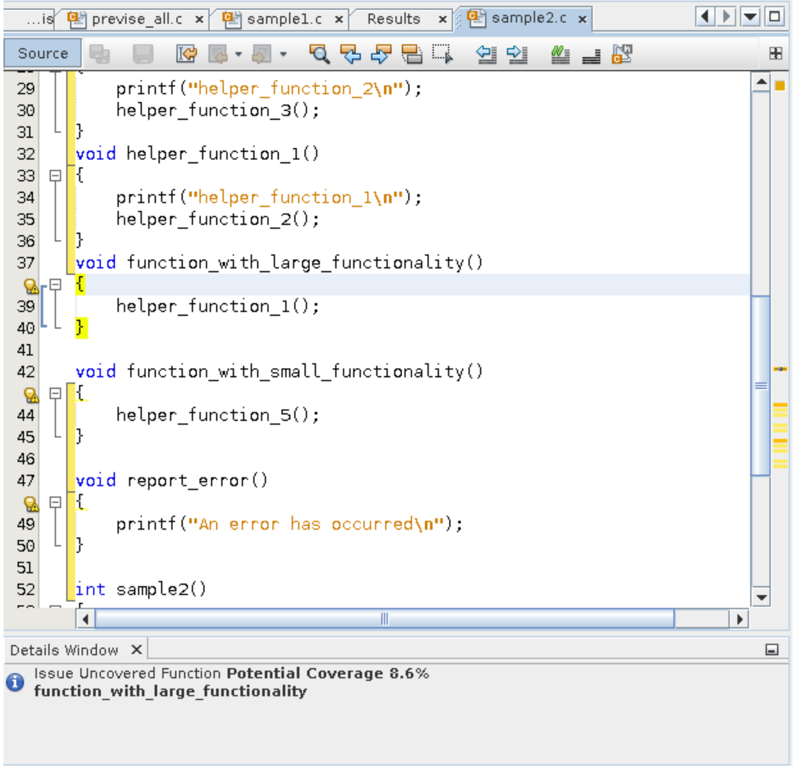


```
Uncovered Function: Potential Coverage 8.6%
function_with_large_functionality
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample2.c:38
35:     helper_function_2();
36: }
37: void function_with_large_functionality()
38: {
39:     helper_function_1();

Uncovered Function: Potential Coverage 2.2%
function_with_small_functionality
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample2.c:4
39:     helper_function_1();
40: }
42: void function_with_small_functionality()
43: {
44:     helper_function_5();

Uncovered Function: Potential Coverage 1.3%
report_error
[redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample2.c:4
44:     helper_function_5();
45: }
47: void report_error()
48: {
49:     printf("An error has occurred\n");
```

8. Open the source file by clicking the source file path link in one of issues. Scroll down in the source file until you see the warning icons in the left margin.



```
    printf("helper_function_2\n");
    helper_function_3();
}
void helper_function_1()
{
    printf("helper_function_1\n");
    helper_function_2();
}
void function_with_large_functionality()
{
    helper_function_1();
}

void function_with_small_functionality()
{
    helper_function_5();
}

void report_error()
{
    printf("An error has occurred\n");
}

int sample2()
```

Issue Uncovered Function **Potential Coverage 8.6%**
function_with_large_functionality

Code that is not covered is marked with a yellow bracket.

The coverage issues found in the file are marked with warning icons ⚠.

Using the `codean` Command-Line Tool

You can similarly use all of the features in Code Analyzer with the `codean` command . This section is a short tutorial on how to use the `codean` command to catch new static code issues in your code, using the same `sample` program from `SolarisStudioSampleApplications`.

1. Previous sections of this tutorial did not compile the `sample4.c`. Preview this file with the `cat` command.

   ```
   $ cat sample_4.c
   int another_new_umr()
   {
     int i;
     if (i)
       return 0;
     else
       return 1;
   }
   ```

   Notice that `int i` is uninitialized.
2. Compile the source and generate the static report.

   On Oracle Solaris:

   ```
   $ cc -g -xprevise main.c previse_1.c previse_all.c sample1.c sample2.c sample3.c
   ```

   On Oracle Linux:

   ```
   $ cc -xannotate -g -xprevise main.c previse_1.c previse_all.c sample1.c sample2.c sample3.c
   ```
3. Save the static report using the `codean --save` option to `a.out`.

   ```
   $ codean --save -s a.out
   ```
4. Recompile the `sample` application, this time including `sample4.c`.

   On Oracle Solaris:

   ```
   $ cc -g -xprevise *.c
   ```

   On Oracle Linux:

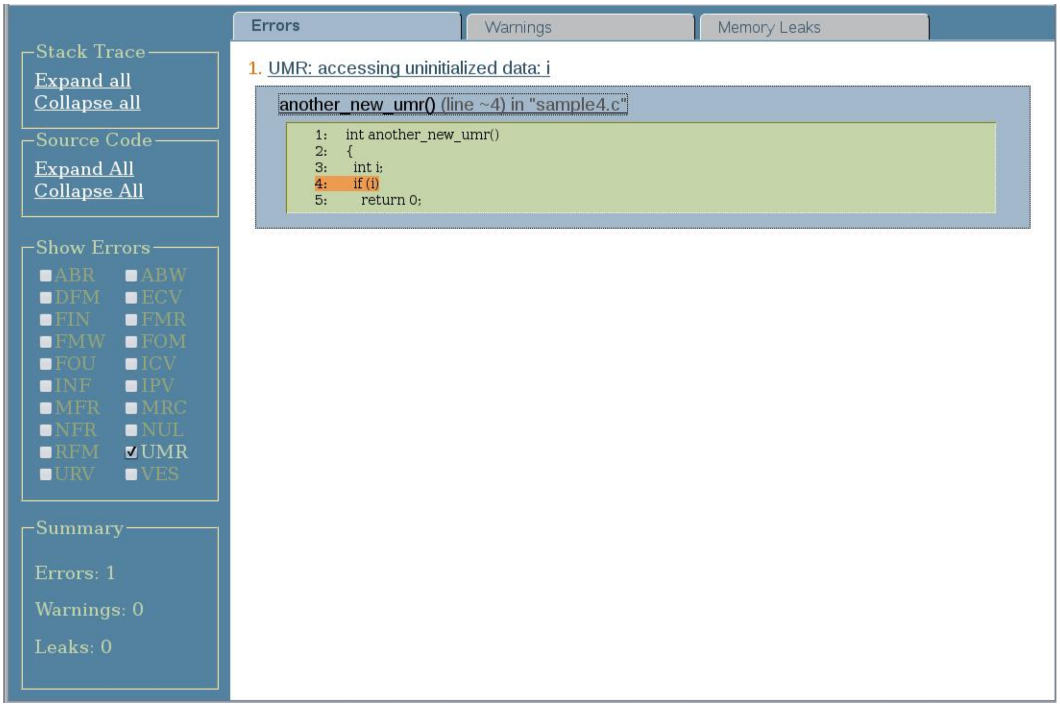
   ```
   $ cc -g -xannotate -xprevise *.c
   ```

   This new function is never called from `main`,however it will introduce a new UMR error.
5. Use the `--whatisnew` option to get a report on the newly added static issue.

   ```
   $ codean --whatisnew -s a.out
   STATIC report of a.out showing new issues:
   Compare the latest results against a.out.analyze/history/2014.8.4.14.49.56...
   ERROR 1 (UMR): accessing uninitialized data: i at:
           another_new_umr()  <sample_4.c : 4>
                   1:      int another_new_umr()
                   2:      {
                   3:        int i;
                   4:=>      if (i)
   ```

```
                 5:          return 0;
PREVISE SUMMARY for a.out: 1 new error(s), 0 new warning(s), 0 new leak(s) in total
```

The following figure shows the HTML report on static code issues generated by codean.



1. UMR: accessing uninitialized data: i

another_new_umr() (line ~4) in "sample4.c"

```
int another_new_umr()
{
  int i;
  if (i)
    return 0;
```

Errors: 1

Warnings: 0

Leaks: 0

For more information about codean, see “Code Analyzer Command-Line Interface” in “Oracle Solaris Studio 12.4: Code Analyzer User’s Guide ” and the codean(1) man page.

Using the Issues Found by Code Analyzer to Improve Your Code

By fixing the core issues found by Code Analyzer, you should be able to eliminate many other issues found in your code, and make major improvements in its quality and stability.

Although static error checking can find the risky code in your application, it can also generate false positives. Dynamic checking can help verify and eliminate these errors, providing a more accurate picture of the issues in your code. Code coverage checking can also help you improve your dynamic test suites.

Code Analyzer integrates the results of these three types of checking to give you the most accurate analysis of your code all in one tool.

Potential Errors Found by Code Analyzer Tools

The compilers, discover, and uncover find static code issues, dynamic memory access issues, and coverage issues in your code. This section lists the specific error types that are found by these tools and analyzed by Code Analyzer.

For more information about these errors and warnings, see Appendix A, “Errors Analyzed by Code Analyzer,” in “Oracle Solaris Studio 12.4: Code Analyzer User’s Guide ”.

Static Code Issues

Static code checking finds the following types of errors:

- ABR: beyond array bounds read
- ABW: beyond array bounds write
- DFM: double freeing memory
- ECV: explicit type cast violation
- FMR: freed memory read
- FMW: freed memory write
- INF: infinite empty loop
- MLK: memory leak
- MFR: missing function return
- MRC: missing malloc return value check
- NFR: uninitialized function return
- NUL: null pointer dereference, leaky pointer check
- RFM: return freed memory
- UMR: uninitialized memory read, uninitialized memory read bit operation
- URV: unused return value
- VES: out-of-scope local variable usage

Dynamic Memory Access Issues

Dynamic memory access checking finds the following types of errors:

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

Dynamic memory access checking finds the following types of warnings:

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

Code Coverage Issues

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