Dynamic Memory Solutions

Leak Check®
Version 2.1 for Linux™

User's Guide

Including Leak Analyzer

For x86 Servers

Document Number DLC20-L-021-1

www.dynamic-memory.com

Notices

Information in this document is subject to change without notice.

Trademarks

Dynamic Memory Solutions, Dynamic Leak Check, Leak Check and Leak Analyzer are trademarks of Dynamic Memory Solutions LLC.

Linux is a registered trademark of Linus Torvalds in several countries.

All other trademarks are the property of their respective owners.

Comments

Comments about this document may be sent to:

Dynamic Memory Solutions LLC
Publications Department
271 Kings Highway
North Haven, CT 06473

Voice/Fax: 1-877-293-4144
Email: *sales@dynamic-memory.com*
Web: *www.dynamic-memory.com*

Copyright

Contents

Preface

Dynamic Memory Solutions

From Day One, we have focused our efforts on providing you with the highest quality software development tools to enable you to maximize your productivity and the quality and stability of your code. We backup our products with the level of technical support that you would expect from experienced software professionals that have been developers themselves. Our goal is to help you use our products in the most effective way possible and we welcome your opinions and suggestions.

You can contact DMS via:

Our Website

www.dynamic-memory.com

Email

Sales: sales@dynamic-memory.com
Support: support@dynamic-memory.com

Telephone

Voice: 1-877-293-4144 (8am – 5pm EST)
Fax: 1-877-293-4144

Support

DMS is fully committed to supporting our products. With each sale, DMS establishes a support agreement where technical support is provided via one or more of the following:

- ***Website -*** Please visit our website at www.dynamic-memory.com for answers to the most frequently asked questions regarding installation and use. A recent version of the frequently asked questions is also included in this manual.

- ***Email -*** support@dynamic-memory.com -- We attempt to answer email questions within three business days though most will be answered by the next business day.

- ***Telephone -*** Telephone support is obtained by calling 1-877-293-4144 during our regular business hours (8am – 5pm EST). Your support agreement may include telephone support. Otherwise, it is provided at additional cost.

- ***On-site support -*** DMS will travel to your site or connect to your server via an externally accessible network. Our consultants will help diagnose and resolve difficult problems found while using our tools. On-site support is negotiated on a case-by-case basis. Please call our Telephone Support line for more information on this service.

- ***Training –*** DMS will arrange training seminars via electronic delivery or in person at your facility. Please contact us for additional information.

Leak Check Functional Overview

Leak Check™ is an advanced memory-leak detection tool. It checks all code within a process including shared libraries and third party code for a variety of memory management errors. As with all products in the Dynamic Suite, Leak Check requires no modification of the application to insert compile-time or link-time instrumentation.

The major features of Leak Check include:

- ***C and C++ support*** – All features work with either programming language.
- ***Memory leak detection*** – All memory allocations are carefully tracked. If the program loses all references to a particular allocation and therefore 'leaks', Leak Check reports the problem.
- ***Multiple de-allocation*** – Calling delete or free more than once for the same allocated block is a serious error that often causes a program to crash. Leak Check blocks additional deletes or frees and reports the error.
- ***Memory block over/underwrite*** (bounds checking) – Heap allocated blocks are subject to being written beyond the beginning or ending boundary under a variety of conditions. This can cause memory corruption that causes the program to behave erratically or crash. Leak Check detects and pinpoints the over/underwriting in most cases and reports the error.
- ***Other memory errors*** - A variety of other errors are also identified including Free without Malloc, Realloc without Malloc, Free then Realloc and more.
- ***Forked program checking*** - Leak checking includes reporting of leaks in forked child programs.
- ***32/64 bit and multithread support*** -- All features work in both 32 and 64 bit memory models and multi-threaded applications.
- ***Periodic results updating*** – User can configure leak check reporting to occur at an interval of their choice. This is particularly useful for long-running applications.
- ***Graphical User Interface*** – Results can be viewed and manipulated using the Leak Analyzer GUI for easier understanding. Requires X11 support.

The capabilities of Leak Check enable your organization to locate and correct program problems quickly and easily. Since no instrumentation is required, the tool is perfect for use in production environments as well as test and development environments.

Installing Leak Check

Installation requirements

Minimum system requirements for Leak Check:

- A Linux distribution with kernel version 2.4.20 and greater or 2.6.4 and greater on an i586 platform. Please contact us for current availability on other kernels and platforms.
- Linux libelf package installed (version 0.8.4 recommended). This package in included in most Linux distributions and may also be downloaded from the Internet.
- The graphical user interface, Leak Analyzer, requires a working X11 installation. However, Leak Analyzer does not have to be run on the same server that is used for running Leak Check.
- At least 50 MB of available disk space

Obtaining the Leak Check software

The first step is to read and accept the Leak Check software license. If you do not accept our license, you should not download our software and should remove any existing copies immediately.

Please contact us to obtain a copy of Leak Check. We prefer electronic distribution via our website or email but special arrangements can be made for delivery on Compact Disc. Call us at 1-877-293-4144 or email us at sales@dynamic-memory.com.

When you contact us, we will request information from you regarding the system you will use to host our software. This information permits us to generate a license key to permit our software to run on that server or workstation. Your license key options are:

- **Evaluation license** – An evaluation license key enables full program operation for a short time to allow evaluation of Leak Check. There is no charge for this type of license.
- **Full license** – A full license key enables you to use Leak Check on the designated server or workstation and entitles you to minor program updates. Normally this entitles you to use the software for an unlimited period of time as long as you abide by the license.

Installation procedure

Dynamic Memory Solutions offers a number of products to maximize your productivity and software quality. Normally, you will want to install these products in the same directory. Installation of the Leak Check software does not modify any system files and while installation as root is suggested, it is not required but facilitates use by multiple users. Our products can be installed on a shared file system but you will still require a license key for each client machine.

Leak Check is distributed as a compressed tar file (tar.gz) with a top-level directory named dms-2.0/. The complete directory structure is:

dms-2.0/bin	executable binaries
/lib	supporting libraries
/env	environment setup scripts
/license	product license files
/docs	product documentation
/examples	example program and results
/extras	optional or supporting files

Before you install

In preparation for installation please follow the following steps:

- Obtain the software directly from Dynamic Memory Solutions.
- Obtain either an evaluation or full license key. To provide this key, we require information about the system on which our software will be running. The information we require is purely hardware related and requires no disclosure of software or data on your server.

  To obtain the required information, issue the three commands **hostname, ifconfig** and **uname -a** as either root or user and record the values. Additionally, please provide the processor brand and model and the total number of CPU cores.

  For example, the following commands will capture the information to a file that can be sent to us for key generation:

```
hostname  >  /tmp/<descriptive name>.dmshost
ifconfig  >>  /tmp/<descriptive name>.dmshost
uname -a >>  /tmp/<descriptive name>.dmshost
```

Then either email this information or this file to sales@dynamic-memory.com or print it and fax it to us at 1-877-293-4144. Please use a descriptive name that identifies the system to you and we will reference this name when we send you the license key. When you request multiple license keys, this allows you to match the key to the corresponding system.

First install

If you are installing your first DMS product, follow these steps to install the Leak Check software.

- Decide where you will install Leak Check and create the directory if it does not exist. For example, directory /apps. Ensure that users of the product will have read access to this directory. You may also install the product to a shared file system (e.g. NFS).

- Change to the directory chosen for the install (e.g. /apps). Uncompress and install the product with the following commands:

```
cd /«install directory»
cp «full path / product filename»  .
tar –xvzf  «product filename»
```

  This results in the creation of the dms-2.0/ directory and subdirectories containing the product code. For example, if you installed in the /apps directory, directory /apps/dms-2.0 is created. This is your DYNROOT directory and you set the environment variable DYNROOT to it.

- Copy the Leak Check license file (if sent separately) to a directory where users have read permission. The default location for the license file is $DYNROOT/license/ . License keys may be kept on a shared file system if that is where you installed the software.

  If you choose a license key file directory and name other than the default $DYNROOT/license/license.dat, the $DYNLICENSEFILE environment variable must be set to the full path and name of the file before you run the software.

Additional product install

If you have other Dynamic Memory Solutions products installed, it is highly recommended that you install Leak Check in the same location. This allows users to access any of the products at the same release level without needing to modify environment variables including $DYNROOT, $PATH and $LD_LIBRARY_PATH when switching between products.

When installing a new release of a DMS product, the name of the top-level directory in the archive is different in order to support multiple installed releases. For example, if you have already installed Leak Check release 1.0 in the /apps directory, the product is in directory /apps/dms-1.0. When you install Leak Check release 2.0, it is installed in directory /apps/dms-2.0. You can use either version by setting your $DYNROOT environment variable to the corresponding directory.

User configuration

Before running Leak Check, you must configure the environment by setting a few variables. The DYNROOT environment variable must be set to the directory containing the release (e.g. /apps/dms-2.0). Once this is set, you run a script we provide (dms-2.0/env/dyn.env) to set the other variables PATH and LD_LIBRARY_PATH.

```
export  DYNROOT=«install directory»
$DYNROOT/env/dyn.env
```

The above commands may be added to a user's shell profile. For example, add these

lines to the .kshrc file to configure the variables and Leak Check is available by simply typing its name.

Quick Start

Leak Check is easy to use and comes with default settings appropriate for most C and C++ applications. This section describes how to use the basic features of Leak Check on a program.

To use Leak Check:

1. Compile and link your program as you normally do. It is not necessary to compile with the debug flag, but Leak Check can provide more information if the application is compiled with the debug flag enabled.

2. Ensure that the $DYNROOT environment variable is set and that the $DYNROOT/bin directory is in your $PATH. Set the $DYNOUTPUT environment variable to the directory where you want Leak Check to write the results.

3. Execute

   ```
   dynleak «program name»  «program options»
   ```

   where «program name» is the name of your executable program, and «program options» are the command line parameters you use for your program.

4. When your program has completed, view the report files in the $DYNOUTPUT directory. If you did not set $DYNOUTPUT or you do not have write permissions to that directory, the output files will be created in the /tmp directory. The report files contain the call stack for any allocations leaked by your program. Three files may be created:

   - ***programname.pid.dlcsum*** – A textfile containing a summary of the memory allocations and deallocations during the program execution. Included is a summary of memory leaks and boundary errors.

   - ***programname.pid.dlcdet*** – A textfile containing detailed locations of errors including allocations leaked, freed but not allocated and boundary errors. This file is only created when memory errors are found.

   - ***programname.pid.dlcbin*** – A binary file that is used by the graphical user interface program Leak Analyzer.

Feature Descriptions

Allocated memory use error detection

Leak Check monitors the allocation and deallocation of memory blocks and detects when the application attempts to free blocks that have already been freed or when the application passes an invalid pointer to free. The erroneous free is detected, prevented and reported. Leak Check also reports the first and successful free if the DYNRECORDFREESTACKS environment variable is set prior to running the application.

Memory leak detection

This feature provides fundamental memory leak information. Leak Check monitors all memory allocations, and reports any outstanding heap allocation without a reference as a leak. The Leak Check report includes the call stack and line number where the program allocated the leaked memory. If a source line leaks multiple times, Leak Check reports the total number of leaked allocations and the total number of bytes leaked.

Potential memory leak detection

When the only reference to a heap allocation is a reference to a memory address other than the beginning of the block, Leak Check reports this block as a potential leak. In order for the program to deallocate the block, or free the memory, the program must know the initial address of the allocation, so it is likely that the potentially leaked allocation is actually leaked.

Freed memory access checking

The Freed Memory Access feature can help detect program errors when the program attempts to access previously deallocated memory. Using freed memory is a serious error. To help catch such errors, Leak Check can fill each block with zeros when free or delete is called. Clearing memory often causes a program to crash when freed memory is accessed, rather than allowing the program to continue with the invalid data. The core file can then be used to determine the location where the program accessed the invalid memory.

Retention of freed memory

Memory corruption errors may cause a program to crash. Since Leak Check shares heap space with the application, a program with bugs may crash unexpectedly even when Leak Check is being used. The No Free option prevents the return of memory to the system when free or delete is called. In this fashion, each block of memory is only used once. Setting this option may keep a program with memory corruption errors from crashing and allow the program to be debugged.

When this feature is used, the memory usage of a program will continually increase. **The program must end before exhausting available heap space.**

Allocated block boundary checking

Writing beyond the boundaries of an allocation is a serious memory corruption error. By default, Leak Check validates each block for underflow (writing before the start of the allocated block) and overflow (writing after the end of the allocated block). This problem often occurs when an array index goes beyond the defined bounds of the array.

Boundary checking is performed when the user's program exits. , The user can also use the DYNBOUNDARYCHECKCOUNT option to perform boundary checking every time the configured number of allocations have been made.

Multiple reporting options

Leak Check results are generated when your program exits. Results can also be updated while the user's program is is running. The DYNLEAKTIMER option allows the user to request periodic updating of results. The user may also signal Leak Check at any time to cause an update.

Using Leak Check

Setup

Before using Leak Check, it must be installed on your server with a valid license key. Please see the product installation section for more information.

In order to use Leak Check, the $DYNROOT environment variable must be set to the directory chosen for the product at installation. Additionally, you will need to add the $DYNROOT/bin directory to your $PATH environment variable. The setting of these variables is independent of the location of your application program and may be included in your profile for increased ease of use.

Output from Leak Check is directed to the directory specified by the $DYNOUTPUT environment variable. In a development environment, it may be most convenient to set this to the directory used to compile and link the executable. In a test environment where the executable is already built and resides in a shared, read-only directory, $DYNOUTPUT may be set to a directory owned by the tester with write access. If $DYNOUTPUT is not set or is set to a directory to which the user does not have write access, output will be directed to the */tmp* directory.

Invoking Leak Check

The application program is compiled and linked in your normal manner. It is not necessary to compile the application with the debug flag, but Leak Check can provide more information with the debug flag enabled.

Set any additional Leak Check options required for your test. Options are identified in the following section and are set using environment variables.

After performing these steps, invoke Leak Check as follows:

```
dynleak  «program name»  «program options»
```

where «program name» is the name of your executable program, and «program options» are the command line parameters you would normally use for your program.

Options

Leak Check supports a number of options. These options are enabled by setting the environment variable indicated by the option name to the desired value. They are disabled by unsetting the environment variable. For example, to prevent the writing of banner information and results summary to standard output, execute the statement found on the top of the next page:

```
export DYNQUIET=1
```

To re enable the writing of banner information and results summary to standard output after it has been disabled, execute this statement:

```
unset DYNQUIET
```

The following options are supported:

DYNROOT = [directory]

The DYNROOT variable must be set in the user's environment. It contains the Leak Check installation directory.

DYNLICENSEFILE = [path and file name]

This variable overrides the default path to the license file. If the license file is not located in the default directory, then this option designates its location including full path and file name.

Default value is $DYNROOT/license/license.dat

DYNPATH = [path]

This option designates an alternate path that Leak Check uses to locate the application source files. Access to the source files allows Leak Check to provide more complete information about the memory errors within a program.

Multiple paths can be specified by delimiting their values with a colon (:) within the value of $DYNPATH. For example:

```
DYNPATH=/myproduct/libs/libsrcdir:/myproduct/src/programsrcdir
```

If this variable is not specified, the path included by the compiler in the debug information within the object file will be used.

Default value is *not set.*

DYNOUTPUT = [directory]

The DYNOUTPUT variable specifies the directory where Leak Check writes the output files. If the DYNOUTPUT option is not specified, or designates a directory where the user does not have write permission, then the output files are written to the /tmp directory.

Default value is *not set.*

DYNQUIET

Leak Check writes banner information and the results summary to standard output. Setting the **DYNQUIET** environment variable prevents this writing to standard output (stdout). This is useful in a variety of situations including program I/O from the stdin/stdout, piping program output to another program or running the target program in the background.

Set this variable to prevent banner information from being written to standard output. If this variable is not set, banner information is written.

Default value is *not set.*

DYNRECORDFREESTACKS

Set the DYNRECORDFREESTACKS option to cause Leak Check to report successful memory deallocations when they are followed by erroneous deallocations of the same memory block. If this option is not set, Leak Check reports details of the erroneous deallocation.

Default value is *not set.*

DYNFREEMEMORYCLEAR

Set the DYNFREEMEMORYCLEAR option to cause Leak Check to file the allocated blocks with zeros when they are deleted or freed. If this option is not set, Leak Check does not clear blocks when they are deallocated.

Default value is *not set.*

DYNNOFREE

The **DYNNOFREE** option prevents Leak Check from returning memory to the system when it is freed or deleted. This may prevent application crashes until the problem is corrected.

Warning: When this option is used, the memory usage of a program continually increases. **The program must end before running out of memory.**

Setting this variable prevents freed memory from being returned to the heap. If this variable is not set, Leak Check allows the normal return of allocated memory.

Default value is *not set.*

DYNNOBOUNDARYCHECK

The No Boundary Check option controls allocated memory block boundary checking.

Set this variable to prevent memory block boundary checking. If this variable is not set, Leak Check performs boundary checking.

Default value is *not set.*

DYNBOUNDARYCHECKCOUNT = [count]

The Boundary Check Count option is set to cause Leak Check to perform boundary over/underrun checking every time the specified number (count) of memory allocations take place. Using this option and monitoring the output files may provide the user with a better idea of where in the application memory block over/underruns are occurring.

Set this variable to the number of memory allocations to occur before performing boundary error checking again. If not set, boundary error checking occurs only at program exit. ***Warning:*** Do not set the value too low or program thrashing will occur.

Default value is *not set.*

DYNLEAKTIMER = [seconds]

The Leak Timer option is set to cause Leak Check to update and report results based on elapsed time. This option is particularly useful for long running applications. Setting this variable allows the user to monitor the program execution for leaks and errors and to establish trends based on elapsed time or associated with processing loads and data sets.

Note: the report is not updated when the timer expires until the program performs a memory allocation or deallocation.

Set this variable to the number of seconds of wall time between results update. If not set, results are generated only at program exit. ***Warning:*** Do not set the value too low or program thrashing will occur.

Default value is *not set.*

DYNNOPOTENTIALLEAK

The No Potential Leak option allows the user to suppress reporting of potential memory leaks. This may result in a smaller report file when the user is not interested in potential memory leaks.

Set this variable to prevent potential memory leak reporting. If this variable is not set, Leak Check includes potential memory leaks in the report.

Default value is *not set.*

DYNNOFORK

The No Fork option allows the user to control leak checking of child processes. This may result in a smaller report file and faster execution when the user is not interested in processes forked by the main program.

Set this variable to prevent child processes from being checked. If this variable is not set, Leak Check will check child processes for memory leaks.

Default value is *not set.*

Results

Leak Check reports results when the user's application exits. Additionally, the user can configure results to be generated periodically during program execution or at any time via a signal. Result files are written to the $DYNOUTPUT directory. If the $DYNOUTPUT environment variable is not set or you do not have write permission for that directory, the output files are created in the /tmp directory. Three files may be created:

- ***programname.pid.dlcsum*** – A text file containing a summary of the memory allocations and deallocations during the program execution. This file provides a summary of memory leaks and boundary errors.

- ***programname.pid.dlcdet*** – A text file containing detailed locations of errors including allocations leaked, freed but not allocated and boundary errors. This file is only created when memory errors are found.

- ***programname.pid.dlcbin*** – A binary file used by the graphical user interface program Leak Analyzer.

Updating results periodically

The DYNLEAKTIMER environment variable is used to cause Leak Check to update and report results based on elapsed time. Please refer to the DYNLEAKTIMER description in the Options section for complete details.

Updating results on demand

Leak Check monitors two operating system signals and the user may trigger a results update by sending one of these signals to the application process. These signals are SIGUSR1 and SIGUSR2.

If the user's program has a signal handler for one or both of these signals, then the signal cannot be used to trigger the results update. Usually programs do not use the SIGUSR1 and SIGUSR2 signals.

Excluding known leaks from reports

By default, Leak Check reports memory leak information for all libraries, source files and functions in or used by the program, including the standard C and C++ libraries. There may be known memory leaks in these functions and libraries that cannot be corrected and appear in the reports.

Leak Check provides the ability to exclude certain functions or entire libraries from the leak check reports.

Functions to be excluded are entered in the file **dynleak.function.excl**

dynleak.function.excl is a simple text file with one function name per line. A function exclusion file looks like this:

```
myFunction
strcpy
```

Libraries to be excluded are entered in the file **dynleak.library.excl**. Excluding libraries can speed up Leak Check initialization. Since debug information in excluded libraries is not read, any defects reported in these libraries do not contain line or file information.

dynleak.library.excl is a simple text file with one shared library name per line. Library paths are not required. A library exclusion file looks like this:

```
mySharedLibrary.so
libncurses.so.5
```

These files can be placed in either the user's HOME directory or in the directory $DYNROOT/env

Exclusion files in the two directories are joined to create a complete list of functions and libraries to exclude from the reports.

The files in $DYNROOT/env are typically used to list standard C and C++ libraries, or other 3rd party libraries linked into the application and the files in the HOME directory are used by individual developers.

Libraries and functions to be excluded must be added to the exclusion file prior to running Leak Check**.**

Understanding the results

When run, Leak Check generates two or three output files. The summary output file is always generated and is named **«program name».«PID».dlcsum**. **«program name»** is the name of the user's executable and **«PID»** is the Process ID of this execution of the program. If there are reportable errors, a detail output file is generated and is named **«program name».«PID».dlcdet**. As with the summary output file, the **«program name»** is the name of the user's executable and **«PID»** is the Process ID of this execution of the program. The third file is named **«program name».«PID».dlcbin**. This file is used by the graphical user interface Leak Analyzer.

Summary output file

The summary output file contains an overview of the results from Leak Check. Only the total number of each error type and associated number of bytes are provided in this file.

This following is an example of a summary output file:

```
Dynamic Leak Check Summary

1 → Critical errors                    4
2 → Boundary Overflows                 1
    Bytes leaked                    3020
3 → Blocks leaked                      5
    Unique stacks for leaks            5
    Bytes potentially leaked        2000
4 → Blocks potentially leaked          1
    Unique potential leak stacks       1
    Bytes suppressed                34411
5 → Blocks suppressed                 27
    Unique suppressed stacks          22
6 → Bytes in use                    39463
    Blocks in use                      34
7 → Peak bytes in use               43276
```

"Critical errors" (**1**) are memory errors that could cause a program to crash. These include free without malloc, freeing a pointer to the middle of a block, and multiple free. The total count of critical errors is provided here.

"Boundary overflows" (**2**) indicates the number blocks where the program has written beyond the boundaries of allocated memory. If the program writes before the start of a block or beyond the end of a block, the value is incremented by one. If the program writes before and beyond the block, the value is incremented by two.

"Bytes leaked", "Blocks leaked" and "Unique stacks for leaks" (**3**) indicate the total number of bytes and the number of allocations of heap memory leaked during program execution. Additionally, the number of unique stacks provides the user with an indication as to how many distinct leaks are in the executable program. For example, if the only leak in a program is in a loop that is executed 10 times, there are 10 blocks leaked but the number of unique stacks is one.

“Bytes potentially leaked”, “Blocks potentially leaked” and “Unique potential leak stacks” (**4**) indicate the total number of bytes and the number of allocations of heap memory where the program has not maintained a pointer to the start of the allocation throughout program execution. However, for each, there is a pointer to a location within the block so the user could, potentially, return it. This pointer may provide access to a data structure within the block but should be examined for correctness. The definition for “Unique potential leak stacks” is the same as for “Unique stacks for leaks”. It is an indication as to the number of distinct potential leaks in the code.

“Bytes suppressed”, “Blocks suppressed” and “Unique suppressed stacks” (**5**) indicate the total number of bytes and the number of allocations of heap memory leaked or potentially leaked during program execution but not reported because the user has chosen to suppress reporting. A user may choose to suppress reporting of known leaks that occur in library functions or other system or application functions that cannot be fixed to streamline reporting via the exclusion files. The number of unique stacks is reported to provide an indication as to how many distinct leaks or potential leaks are in the code but are suppressed.

“Bytes in use” and “Blocks in use” (**6**) indicate the total number of bytes and the number of allocations of heap memory still in use by the executable (and not returned) at program termination. While these blocks may no longer be needed, the program has maintained references so they are not considered leaks even though they are not returned. These values may include allocations created by the compiler to support program execution. This memory is recovered by the operating system during program termination.

“Peak bytes in use” (**7**) indicates the greatest number of bytes allocated during this program execution at any point in time. It identifies the minimum amount of dynamically allocated memory require by the program under these specific execution conditions and this input dataset.

Detail output file

The detail output file contains information about each reportable memory management error or memory leak detected by Leak Check during program execution. There are four classes of errors: critical errors, boundary errors, memory leaks, and potential memory leaks.

Critical errors are reported first, followed by boundary errors and then memory leaks including potential leaks. Leaks are sorted so the largest and most frequent leaks appear first in the report. Multiple leaks from the same execution stack are combined into a single report item indicating the total number of leaked blocks and the associated number of bytes.

Critical errors

Entries for critical errors provide details for each dynamically allocated memory error detected by Leak Check. Each error includes information on type, address, calling functions and line numbers.

The following allocated memory error types are reportable:

- free within block

- free without malloc
- reallocate without malloc
- double free
- free then reallocate

```
BAD FREE:Identified a bad pointer (0X1234) sent to free
        reportFreeWithoutMalloc+0x120 at example
        free+0x234 at example
        main at line 98 in /tmp/example.c
        _start+0x5c at example
```

The first line of a critical error contains the address of the allocated memory block of a pointer in hexadecimal format and the error type. The following lines provide the call stack leading up to the error. In this example, the error is detected by Leak Check when the the application calls the function **free**. The next line of the call stack shows that **free** is called from line 98 of function **main**. The final line of the call stack is unimportant but shows where the application, **example**, is passed control during application startup.

Errors such as double free, where free is called more than once for the same block, include additional information as shown below.

```
DOUBLE FREE:Identified a block (0X924D18) sent to free more than once.
The allocation call stack:
        main at line 100 in /tmp/example.c
        _start+0x5c at example

The first free call stack was not recorded.
Set the DYNRECORDFREESTACKS environment variable and
run the program again to get both call stacks.

The second free call stack:
        reportMultipleFree+0x130 at example
        free+0x100 at example
        main at line 104 in /tmp/example.c
        _start+0x5c at example
```

When an application attempts to free the same memory block multiple times, Leak Check identifies the location of the initial allocation and the location of the second deallocation of the block. Additionally, if the DYNRECORDFREESTACKS environment variable is set, Leak Check reports the location of the first (and successful) deallocation of that memory block. The call stack information provided is the same as for other errors.

In your detail output file, the function names shown may not actually be used directly by your application but result from explicit or implicit application calls to library functions. Line numbers are only available if debugging information is included in the application and library object files. Pathnames are absolute or relative to the location of the application depending on how the user application is created.

Boundary errors

Entries for boundary errors provide details for each memory block overflow (write past highest memory address allocated) or underflow (write before lowest memory address allocated) detected by Leak Check. When memory is allocated, a guard band is added to

the beginning and end of the block. An error is detected when either of these areas are written. Each error includes information on the boundary error type, address of block, and the allocation call stack.

The following boundary error types may be reported:

- Boundary Overflow
- Boundary Underflow

```
BOUNDARY OVERFLOW discovered after block starting at address 0X924370.
The block was allocated at time 1186160768
The error was reported at time 1186160769
The allocation call stack:
        function2 at line 49 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example
```

The first line of a boundary error contains the address of the allocated memory block in hexadecimal format and indicates whether it is an overflow or underflow. BOUNDARY OVERFLOW indicates that data has been written past the end or highest address of the buffer. BOUNDARY UNDERFLOW indicates that data has been written before the beginning or lowest address of the buffer. The remaining lines provide the call stack leading up to the allocation of the memory block. In this example, the memory is allocated at line 49 of function **function2**. **function2**, in turn, is called from line 95 of **main**. The final line of the call stack is unimportant but shows where the application, **example**, is passed control during application startup.

In your detail output file, the function names shown may not actually be used directly by your application but result from explicit or implicit application calls to library functions. Line numbers are only available if debugging information is included in the application and library object files. Pathnames are absolute or relative to the location of the application depending on how the user application is created.

Memory leak

Memory leaks are reported when the application program no longer has a pointer to the beginning of an allocated memory block. Unlike potential memory leaks, there does not appear to be a pointer to a location within the block that might be used to access the block. Each error includes information on the size of the leak, the address of the block, and the allocation call stack.

```
LEAK:Identified a block of size 500 bytes at address 0X886760
The allocation call stack:
        function1 at line 20 in /tmp/example.c
        main at line 93 in /tmp/example.c
        _start+0x5c at example
```

The first line of a memory leak error provides the size of the leak in bytes and the address of the allocated memory block in hexadecimal format. The remaining lines provide the call stack leading up to the allocation of the memory block. In this example, the memory is

allocated at line 20 of function **function1**. **function1**, in turn, is called from line 93 of **main**. The final line of the call stack is unimportant but shows where the application, **example**, is passed control during application startup.

In your detail output file, the function names shown may not actually be used directly by your application but result from explicit or implicit application calls to library functions. Line numbers are only available if debugging information is included in the application and library object files. Pathnames are absolute or relative to the location of the application depending on how the user application is created.

Potential memory leak

Potential memory leaks are reported when the application program no longer has a pointer to the beginning of the allocated memory block. While it is likely that this is actually a memory leak, the application program may still be using the block and might be able to compute the address of the beginning of the block to return it. Each error includes information on the size of the leak, the address of the block, and the allocation call stack.

```
POTENTIAL LEAK:Identified a single block of size 2000 bytes at address 0X923B88
A word was found pointing at 4 bytes after the beginning of the block
The allocation call stack:
        function1 at line 23 in /tmp/example.c
        main at line 93 in /tmp/example.c
        _start+0x5c at example
```

The first line of a potential memory leak error provides the size of the potential leak in bytes and the address of the allocated memory block in hexadecimal format. The second line indicates the offset of the potential pointer. The remaining lines provide the call stack leading up to the allocation of the memory block. In this example, the memory is allocated at line 23 of function **function1**. **function1**, in turn, is called from line 93 of **main**. The final line of the call stack is unimportant but shows where the application, **example**, is passed control during application startup.

In your detail output file, there may be function names shown that are not actually used directly by your application but result from explicit or implicit application calls to library functions. Line numbers are only available if debugging information is included in the application and library object files. Pathnames are absolute or relative to the location of the application depending on how the user application is created.

Example

Included with the installation of Leak Check is an example program you can use to verify your installation. The example includes the most common types of memory management programming errors and the sample output files show the error messages that Leak Check reports when each error is encountered.

Example files

The example consists of 4 files:

example.c – The source code for the example program

example.1234.dlcsum – A text file containing a summary of the memory allocations and deallocations during the program execution. This file provides a summary of memory leaks and boundary errors.

example.1234.dlcdet – A text file containing detailed locations of errors including allocations leaked, freed but not allocated and boundary errors. This file is only created when memory errors are found.

example.1234.dlcbin – A binary file used by the graphical user interface program Leak Analyzer.

Compiling the example

To compile the example using the gcc compiler, execute this command:

```
gcc -g example.c -o example
```

Source code

The source file, example.c, contains a main function and two supporting functions. Each of these functions contain memory errors. The errors include memory leaks, potential memory leaks, array boundary errors, free without malloc, multiple frees of a memory block and free called with a pointer to the middle of an allocated block.

Dynamic Memory Solutions

This is the text of the source file, example.c:

```
   1    #include <stdlib.h>
   2    #include <stdio.h>
   3    #include <string.h>
   4
   5    int* function1(const int inNbrToMalloc)
   6    {
   7       //Initialize a minimun number to malloc
   8       int nbrToMalloc = 2;
   9       char *myCharArray;
  10       int *myIntArray;
  11
  12       if (inNbrToMalloc > nbrToMalloc)
  13       {
  14          //If the input number is larger than the minimum
  15          //use the input number
  16          nbrToMalloc = inNbrToMalloc;
  17       }
  18
  19       //Allocate an array of chars
  20       myCharArray = (char *) malloc(nbrToMalloc * sizeof(char));
  21
  22       //Allocate an array of ints
  23       myIntArray = (int *) malloc(nbrToMalloc * sizeof(int));
  24
  25       //Advance the pointer to an address within the allocation,
but not at the start
  26       myIntArray+=1;
  27
  28       //Return the pointer to the middle of the array of ints.
  29       //The array of chars is leaked
  30       return myIntArray;
  31    }
  32
  33
  34    int function2(const int inNbrToMalloc)
  35    {
  36       //Initialize a minimun number to malloc
  37       int nbrToMalloc = 2;
  38       char *myCharArray;
  39       int *myIntArray;
  40
  41       if (nbrToMalloc < inNbrToMalloc)
  42       {
  43          //If the input number is larger than the minimum
  44          //use the input number
  45          nbrToMalloc = inNbrToMalloc;
  46       }
  47
  48       //Allocate an array of chars
  49       myCharArray = (char*) malloc(nbrToMalloc * sizeof(char));
  50
  51       //Advance the pointer to an address within the allocation,
but not at the start
  52       myCharArray++;
  53
  54       int i = 0;
  55
```

```
  56       //Write past the end of the allocated array
  57       for (i = 0; i <= nbrToMalloc; i++ )
  58       {
  59          myCharArray[i] = 'A';
  60       }
  61       myCharArray[i] = '\0';
  62
  63       //free the pointer which no longer references the beginning
of the allocation
  64       free (myCharArray);
  65
  66       //Call another function that will allocate memory, and leak
the returned memory
  67       myIntArray = function1(inNbrToMalloc);
  68
  69       return 0;
  70    }
  71
  72    // get rid of any stray pointers on the stack
  73    // In complex programs this is not needed
  74    void clearStack()
  75    {
  76      char bigArray[10000];
  77
  78      memset(bigArray,0,10000);
  79    }
  80
  81
  82    int *myIntPtr;
  83    int main(int argc,char*argv[])
  84    {
  85       int rc = 0;
  86       char *myWildPtr = (char*)0x1234;
  87       char *myDoubleFreePtr;
  88       char myBuffer[100];
  89       char *myBufferPtr;
  90       char *myGoodBuffer;
  91
  92
  93       myIntPtr = function1(500);
  94
  95       rc = function2(400);
  96
  97       //free a pointer which does not reference a block allocated
by malloc
  98       free (myWildPtr);
  99
 100       myDoubleFreePtr = (char *) malloc (50);
 101
 102       free (myDoubleFreePtr);
 103       //free the pointer a second time to cause a multiple free
error
 104       free (myDoubleFreePtr);
 105
 106       //Assign a pointer to an allocation on the stack
 107       myBufferPtr = &myBuffer[0];
 108
 109       //free a pointer to an allocation on the stack
 110       free(myBufferPtr);
 111
```

```
112       //Allocate a buffer that will be properly freed
113       myGoodBuffer = (char *) malloc (120);
114       strcpy(myGoodBuffer, "Example is complete\n");
115
116       printf(myGoodBuffer);
117       clearStack();
118
119       exit(rc);
120    }
121
```

Summary output file

The summary file contains an overview of the results of Leak Check. The text of the summary file for the example program is:

```
Dynamic Leak Check Summary

1 → Critical errors                    4
2 → Boundary Overflows                 1
    Bytes leaked                    2900
3 → Blocks leaked                      4
    Unique stacks for leaks            4
    Bytes potentially leaked        2000
4 → Blocks potentially leaked          1
    Unique potential leak stacks       1
    Bytes suppressed                   0
5 → Blocks suppressed                  0
    Unique suppressed stacks           0
6 → Bytes in use                    5020
    Blocks in use                      6
7 → Peak bytes in use               5020
```

Critical errors (**1**) are memory errors that can cause a program to crash. These include free without malloc, calling free with a pointer to the middle of a block, or multiple frees of the same block. In this example, there are four critical errors.

Boundary overflows (**2**) indicates the number of blocks where the program has written beyond the boundaries of allocated memory. If the program writes before the start of a block or beyond the end of a block, the count is incremented by one. If the program writes before and beyond the block, the count is incremented by two. In the example, there is one boundary overflow.

Bytes and Blocks leaked (**3**) indicate the total number of bytes and the number of allocations of heap memory leaked during the program execution. Additionally, the summary file includes the number of unique stacks where leaks occur. This gives the user an indication as to how many distinct leaks are in the code. If the only leak in a program is in a loop that is executed 10 times, there are 10 blocks leaked but the number of unique stacks is one. The example program leaks 2900 bytes in four memory block allocations from four different locations (stacks).

Bytes and Blocks potentially leaked (**4**) indicate the total number of bytes and the number of allocations of heap memory where the program does not maintain a pointer to the start of the allocation throughout program execution. However, there is a pointer to somewhere within the block so the user can, potentially, access it and return it. This pointer may provide access to a user's data structure within the block but should be examined for correctness. The definition for "Unique potential leak stacks" is the same as for "Unique stacks for leaks". It is an indication as to the number of distinct potential leaks in the code. In this example, 2000 bytes in one block are potentially leaked by the example program from a single location (stack).

Bytes and Blocks suppressed (**5**) indicate the total number of bytes and the number of allocations of heap memory leaked or potentially leaked during program execution but not reported because the user has chosen to suppress reporting. A user may choose to suppress reporting of known leaks that occur in library functions or other system or application functions that cannot be fixed to streamline reporting. The number of unique stacks is reported to provide an indication as to how many distinct leaks or potential leaks are in the code. In this example with a default suppression file, there is no suppression of leak reporting.

Bytes and Blocks in use (**6**) indicate the total number of bytes and the number of allocations of heap memory still in use by the program (and not returned) at the time it terminates. While these blocks may no longer be needed, the program has maintained references so they are not considered leaks even though they are not returned. The example program has 5020 bytes in 6 blocks in use at exit. These may include allocations created by the compiler to support program execution. This memory is recovered by the operating system during program termination.

Peak bytes in use (**7**) indicates the greatest number of bytes allocated during program execution at any point in time. It identifies the minimum amount of dynamically allocated memory required by the program under the specific execution conditions and input dataset.

Detail output file

The detail output file contains information about each memory error or memory leak detected by Leak Check during program execution. Critical errors are reported first, followed by memory overflows and memory leaks. Memory leaks are sorted so the largest and most frequent leaks appear first in the report. Multiple leaks from the same execution stack are combined into a single report item indicating the total number of leaked blocks and the associated number of bytes.

This is the text of the detail output file for the example:

1 →

```
BAD FREE:Identified a bad pointer (0X924371) sent to free
        reportFreeWithoutMalloc+0x120 at example
        free+0x234 at example
        function2 at line 64 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example

BAD FREE:Identified a bad pointer (0X1234) sent to free
        reportFreeWithoutMalloc+0x120 at example
        free+0x234 at example
```

```
        main at line 98 in /tmp/example.c
        _start+0x5c at example

2 →  DOUBLE FREE:Identified a block (0X924D18) sent to free more than once.
The allocation call stack:
        main at line 100 in /tmp/example.c
        _start+0x5c at example

The first free call stack was not recorded.
Set the DYNRECORDFREESTACKS environment variable and
run the program again to get both call stacks.

The second free call stack:
        reportMultipleFree+0x130 at example
        free+0x100 at example
        main at line 104 in /tmp/example.c
        _start+0x5c at example

BAD FREE:Identified a bad pointer (0XFFBEF6B0) sent to free
        reportFreeWithoutMalloc+0x120 at example
        free+0x234 at example
        main at line 110 in /tmp/example.c
        _start+0x5c at example

3 →  BOUNDARY OVERFLOW discovered after block starting at address 0X924370.
The block was allocated at time 1186160768
The error was reported at time 1186160769
The allocation call stack:
        function2 at line 49 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example

4 →  POTENTIAL LEAK:Identified a single block of size 2000 bytes at address 0X923B88
A word was found pointing at 4 bytes after the beginning of the block
The allocation call stack:
        function1 at line 23 in /tmp/example.c
        main at line 93 in /tmp/example.c
        _start+0x5c at example

LEAK:Identified a block of size 1600 bytes at address 0X9246C0
The allocation call stack:
        function1 at line 23 in /tmp/example.c
        function2 at line 67 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example

5 →  LEAK:Identified a block of size 500 bytes at address 0X886760
The allocation call stack:
        function1 at line 20 in /tmp/example.c
        main at line 93 in /tmp/example.c
        _start+0x5c at example

LEAK:Identified a block of size 400 bytes at address 0X924370
The allocation call stack:
        function2 at line 49 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example

LEAK:Identified a block of size 400 bytes at address 0X924518
The allocation call stack:
        function1 at line 20 in /tmp/example.c
        function2 at line 67 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example
```

The first error reported in the detail output file (**1**) is a critical error. All critical errors are reported in a similar manner and we will examine the reporting of this error in detail.

```
BAD FREE:Identified a bad pointer (0X924371) sent to free
        reportFreeWithoutMalloc+0x120 at example
        free+0x234 at example
        function2 at line 64 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example
```

The first line of each error is the error type. In this case (**1**), the error is a BAD FREE or free without malloc. Additionally, the first line contains specific information including address and size as applicable to the error. For this error, it indicates that the application attempted to free a memory block at address `0X924371` that has not been allocated or has been allocated but already freed and returned.

The next lines of the error show the program call stack from the point at which the error is detected, in a Dynamic Memory Solutions library, back to the program invocation. Working from the bottom up, we see where the program **example** is passed control from the operating system. The program executes the *main* function until it reached line 95 where control is passed to *function2*. *function2* executes and at line 64, calls *free*. **dynleak** intercepts the call to free and determines that the memory has not been allocated. This particular error is caused by line 52 of example.c where the pointer to the memory allocation is altered and no longer points to the beginning of the allocation.

A Leak Check v2.0 enhancement is illustrated by the DOUBLE FREE critical error identified by (**2**).

```
DOUBLE FREE:Identified a block (0X924D18) sent to free more than once.
The allocation call stack:
        main at line 100 in /tmp/example.c
        _start+0x5c at example

The first free call stack was not recorded.
Set the DYNRECORDFREESTACKS environment variable and
run the program again to get both call stacks.

The second free call stack:
        reportMultipleFree+0x130 at example
        free+0x100 at example
        main at line 104 in /tmp/example.c
        _start+0x5c at example
```

When an application attempts to free the same memory block multiple times, Leak Check identifies the location of the initial allocation and the location of the second deallocation of the block. Additionally, if the DYNRECORDFREESTACKS environment variable is set, Leak Check reports the location of the first (and successful) deallocation of that memory block.

In this specific example of DOUBLE FREE, the memory block at address 0x924D18 is allocated in *main* at line 100 in example.c. The next part of the error entry indicates that the first deallocation location is not recorded because DYNRECORDFREESTACKS is not set. The final part of the error entry indicates that the second deallocation of the memory

block occurs in *main* at line 104 of the program. If the first deallocation is recorded, it is shown to occur in *main* at line 102.

A BOUNDARY OVERFLOW error (**3**) is reported in a manner similar to a BAD FREE as illustrated by the following part of the detail file.

```
BOUNDARY OVERFLOW discovered after block starting at address 0X924370.
The block was allocated at time 1186160768
The error was reported at time 1186160769
The allocation call stack:
        function2 at line 49 in /tmp/example.c
        main at line 95 in /tmp/example.c
        _start+0x5c at example
```

The first line of the BOUNDARY OVERFLOW error indicates if the overflow is before or after the memory allocation and the address of the allocation. An overflow **after** the block indicates that the program has written to a memory address higher than the allocation. For this error, the program writes beyond the end of the memory block located at address `0X924370`.

The next lines of the error show when the error is detected. The time of each memory allocation, in Unix seconds since epoch, is recorded during allocation of the block. Additionally, the time when the BOUNDARY OVERFLOW is first detected is also provided. Finally, the call stack for the memory block allocation is shown. Working from the bottom up, we see where the program **example** is passed control from the operating system. The program executes the *main* function until it reached line 95 where control is passed to *function2*. *function2* executes and at line 49, it allocates the memory block at address `0X924370`.

POTENTIAL LEAK reporting is illustrated by examining the potential leak (**4**) in the detail output file.

```
POTENTIAL LEAK:Identified a single block of size 2000 bytes at address 0X923B88
A word was found pointing at 4 bytes after the beginning of the block
The allocation call stack:
        function1 at line 23 in /tmp/example.c
        main at line 93 in /tmp/example.c
        _start+0x5c at example
```

The first line of the error indicates that it is a potential leak of 2000 bytes at address `0X923B88`. This memory has been allocated and not freed prior to program exit. However, unlike a leak, a pointer was found pointing 4 bytes after the start of the block. If the program purposely alters the pointer, it is possible that the block is still in use and that it has not been leaked.

The next lines of the error show the program call stack for the memory block allocation. Working from the bottom up, we see where the program **example** is passed control from the operating system. The program executes the *main* function until it reached line 93 where control is passed to *function1*. f*unction1* executes and at line 23 it allocates 2000 bytes of memory via *malloc*.

Examining the source code starting at the memory allocation shows that the pointer to this block is altered at line 26 of example.c.

LEAK reporting is similar as is shown by examining a leak (**5**) reported in the detail output file.

```
LEAK:Identified a block of size 500 bytes at address 0X886760
The allocation call stack:
        function1 at line 20 in /tmp/example.c
        main at line 93 in /tmp/example.c
        _start+0x5c at example
```

The first line of the error indicates that it is a leak of 500 bytes at address `0X886760`. This indicates that the memory has been allocated and not freed prior to program exit. Additionally, no pointer to this allocation is found so it is impossible for the application to free it and, therefore, it is leaked.

The next lines of the error show the program call stack for the allocation. Again, working from the bottom up, we see where the program **example** is passed control from the operating system. The program executes the *main* function until it reached line 93 where control is passed to *function1*. f*unction1* executes and at line 20 it allocates 500 bytes of memory via *malloc*.

Examining the source code starting at the memory allocation shows that a pointer to this memory is neither saved nor returned to the calling function *main*. Since the pointer is lost when *function1* returns, the memory is leaked.

Debugging with Leak Check

Leak Check can be used while you are debugging your program. While in your debugger, you can execute a leak check at any breakpoint. This feature aids you in determining which sections of your code are not properly managing memory resources.

A debugger such as gdb can be used on a process running under Leak Check. All modern Unix debuggers support an 'attach' option. The attach option allows a debugger to take control of a running process. By starting a process under Leak Check and then attaching your debugger, you can combine memory leak checking with source level debugging.

The following steps describe how to attach to a process running under Leak Check.

1. Start your process under Leak Check as usual. Environment variables should be set to direct output and control features.

```
export DYNOUTPUT=/my/output/directory
dynleak example
```

2. Open another terminal window and determine the pid of the process you are debugging with the 'ps' command. In this example, the pid is 1234.

```
ps -ef | grep example

tester  1234  1000  0 02:59 pts/7  00:04:04  dynleak example
```

3. In the second window start gdb with the attach option, specifying the pid of the running process.

```
gdb example 1234
```

The running process is attached and you may use the debugger as normal. Be aware that you may see Leak Check internal functions in the call stacks. These Leak Check internal functions do not impact your debugging session.

You may run a memory leak check at any time from within the debugger. Usually, you will set a breakpoint in your program and allow the process to run until the breakpoint is hit. When your program has stopped at the desired point in gdb, you may run

```
print DYNleakCheck()
```

to execute the memory leak and heap corruption check.

Stopping at a point in your program is very important, if the process is currently running in a Leak Check internal function then running *DYNleakCheck()* may deadlock the process.

The Leak Check output is written to the directory specified by $DYNOUTPUT or to /tmp if $DYNOUTPUT is not set. Only leaks not previously reported will be written.

Leak Analyzer

Introduction

Leak Analyzer is an X Windows based graphical user interface for displaying and analyzing the results from Leak Check, version 2.0 and newer.

Leak Analyzer allows the user to select and load binary formated output files created by Leak Check with an extension of "dlcbin". Once loaded, the user may view information about the user's program that led to the creation of the dlcbin file (Execution Info), the summary results containing the number of each type of error found (Summary Results), the detailed results (Detail Results), and can examine the results in a variety of ways (Leak Analysis). Each of these activities appears on a different tab at the top of the window.

Setup

Leak Analyzer is included with Leak Check. No further installation or setup is required. However, it is X11 Windows based and therefore, you must be able to run X Windows applications. Please ensure that you have the necessary environment variables set and that you can run simple X applications such as xterm prior to running Leak Analyzer.

Invoking Leak Analyzer

After you have executed your application program using Leak Check, you can view the results with Leak Analyzer.

```
dynanalyzer  «options»  «dlcbin file»
```

where **«dlcbin file»** is the name of your executable program, and **«options»** are the command line parameters for Leak Analyzer as described next.

Options

Leak Analyzer has only a few options. These options are passed on the command line when invoking the program and the options can be displayed by passing the “-h” option with no other parameters.

```
Usage: dynanalyzer [-h] [-v] [-display displayname] [<filename>]
  -display display        X server to use (default is $DISPLAY)
  -h                      print a summary of the options
  -v                      print the version number
  <filename>              optionally load dynbin file on startup
```

The “**-h**” option prints the dynanalyzer usage information at the command line. When it is used, all other parameters are ignored.

The "**-v**" option prints the dynanalyzer version at the command line. After printing the version, the program exits.

The "**-display displayname**" option allows the user to override the default X server and to explicitly specify the address and screen number to be used. As an example, the user can specify "-display 111.122.133.144:0" to display dynanalyzer on the X server running at IP address 111.122.133.144 with screen number 0. If this parameter is not specified, dynanalyzer is displayed on the default X server or as specified by the user's DISPLAY environment variable.

The "**<filename>**" parameter allows the user to pass the dlcbin file they want to examine. Leak Analyzer will load this file on startup if it exists. Otherwise, the user can load a file using the program's File menu.

Using Leak Analyzer

Leak Analyzer provides a typical look and feel to other windowed programs. It includes a Menu Bar at the top and a Status Bar at the bottom. Between these two components, a set of tabs allows the user to focus on the specific information they want.

Menu Bar

Top level menu items (File, View, and Help) appear in the menu bar at the top of the Leak Analyzer window.

File

These menu items allow the user to select and load Leak Check binary output files and to close the program.

Open

Open causes the open file dialog to be displayed. This dialog allows the user to browse the filesystem and select a Leak Check binary output file.

When the user has located the desired file, it should be selected (highlighted) and the dialog's ***Open*** button pressed to load the file.

If the user decides not to load a file, the dialog's ***Cancel*** button will close the window and return to the main Leak Analyzer window with the prior contents.

Recent

Recent displays up to 8 of the previously loaded Leak Check binary files in most recently loaded order. A file must have been successfully loaded for it to appear in this list. Older filenames are automatically removed when newer files are loaded.

Files that have been deleted by the user may still appear in the ***Recent*** file list until they are replaced by newly loaded files.

Quit

Quit causes Leak Analyzer to immediately exit.

View

These menu items allow the user to affect the way the information is displayed.

Refresh
Refresh causes Leak Analyzer to reload or refresh the currently loaded Leak Check binary output file.

This menu item is particularly useful when the user has a long running program and is using the periodic reporting capability of Leak Check. Selecting ***Refresh*** causes the current file to be reloaded, thus displaying updated results, if available. The Execution Info tab contains the time when Leak Check last updated the results.

When ***Refresh*** is selected, the data and formating on the Summary Results, Detail Results, and Leak Analysis tabs is reset. If ***Refresh*** is not active, no file is currently loaded that can be reloaded.

Expand All
Expand All causes all items in the table on the current window to be expanded. If ***Expand All*** is not active, no expandable table is currently being displayed.

Collapse All
Collapse All causes all items in the table on the current window to be collapsed. If ***Collapse All*** is not active, no collapsible table is currently being displayed.

Help
These menu items assist the user in operating and understanding Leak Analyzer.

Help Contents
Help Contents opens a new window and displays this help information. Multiple help windows may be open at the same time.

In this window, the user can scroll back and forth to browse the help information or click on a link to go directly to a specific subject. Clicking the ***Contents*** button takes the user to the Table of Contents. Clicking the ***Index*** button takes the user to the Help Index. The ***Back*** and ***Forward*** buttons moves the user back and forth through the links that have been selected. ***Back*** and ***Forward*** does not scroll through the contents. The scroll bar should be used to do this.

To close the **Help Contents** window, select the ***File*** menu from the window and then ***Quit***.

About
About opens a new window containing information about the Leak Analyzer program and how to contact Dynamic Memory Solutions for help with sales and support.

Only one instance of the **About** window may be opened at anytime and it must be closed before the user can continue to use Leak Analyzer.

To close the **About** window, press the ***Close*** button.

What's This?
What's This provides a simple description of any field and answers the question "what's this?"

After selecting this menu item or using the shortcut ***Shift+F1***, the user can click on a field on the screen and receive a short description of the function or purpose of the field. For additional information, the user can consult this detailed help via the Help Contents menu item or the Leak Check User Manual.

Status Bar

Status information regarding program operation is displayed in the status bar at the bottom of the Leak Analyzer window.

Normally, this field displays the status ***Ready***. Ready status indicates that the program is accepting commands.

Other status frequently seen is ***Error loading <filename>*** and ***Try again, file locked by dynleak <filename>***.

The status "Error loading..." indicates that the file opened by the user is not in the correct format. This is usually caused when a file other than a dlcbin file is opened. It may also occur if a dlcbin file becomes corrupted through editing or transfer or if the dlcbin file was generated by a different version of Leak Check.

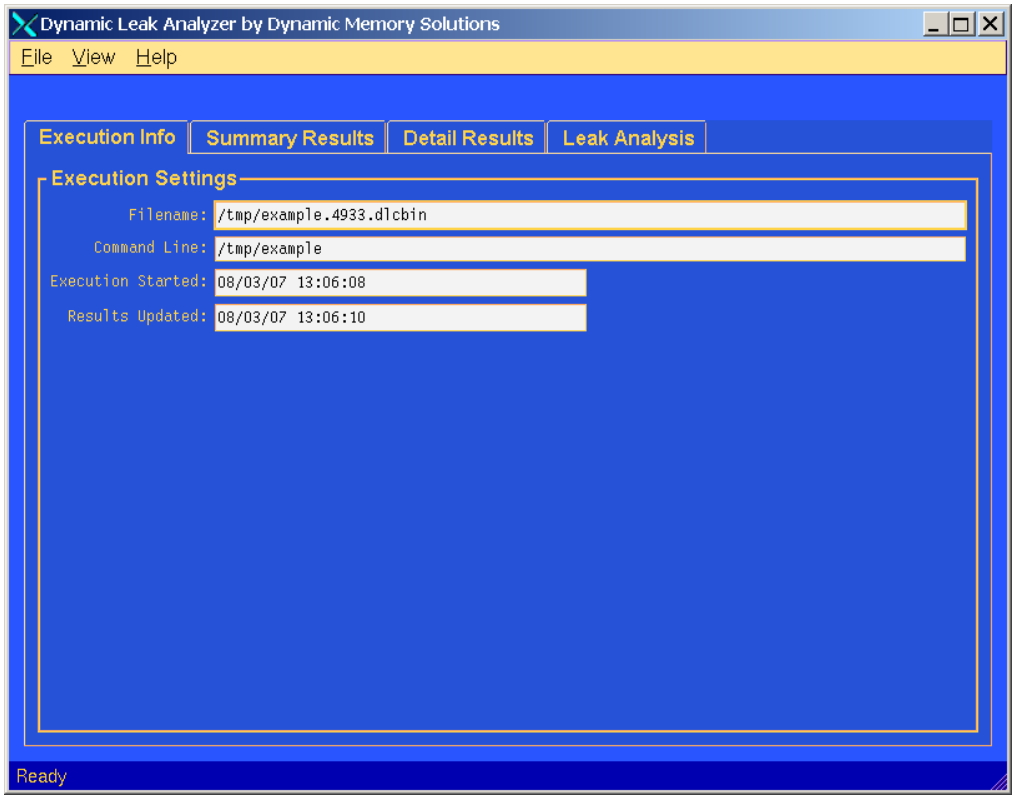
"Try again..." indicates that the dlcbin file is marked as busy. When Leak Check is in the process of writing to the file, it is marked busy. In addition to at program exit, the file is written during periodic updates by Leak Check, if enabled. Once the dlcbin file update is complete, opening or reloading the file should result in success.

Display Function Tabs

The graphical interface provides access to four types of information. Execution Information, Summary Results, Detail Results and Leak Analysis via a tabbed interface.

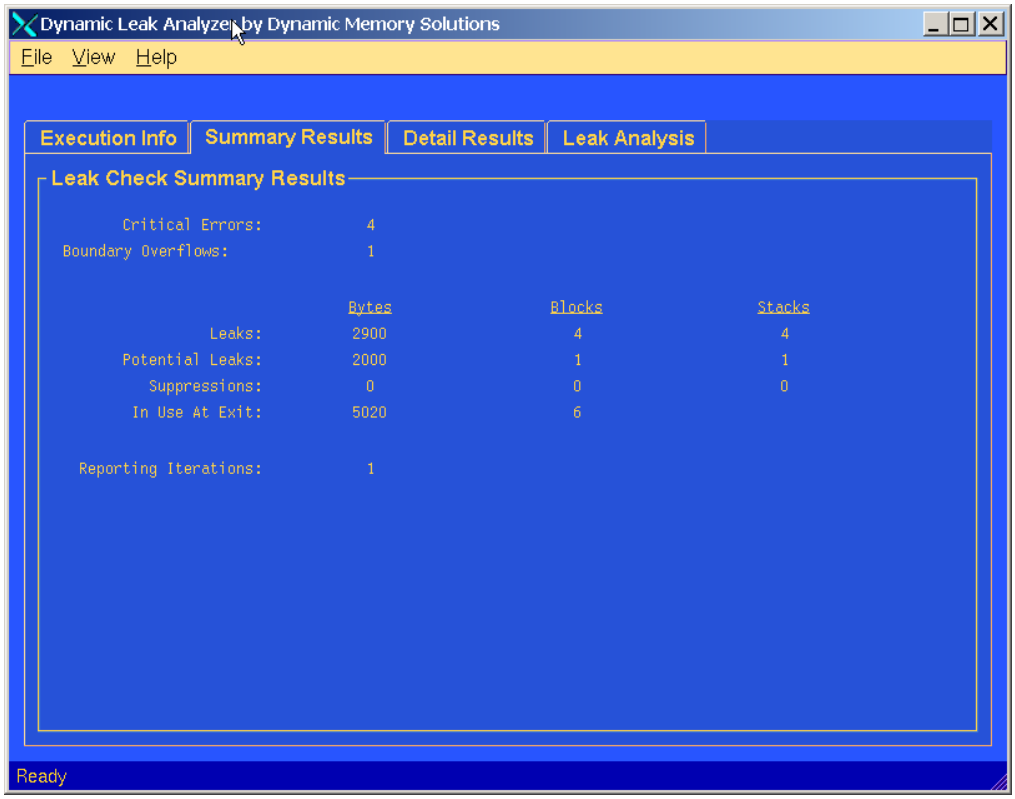
Execution Information

The *Execution Info* tab provides information about currently loaded Leak Check binary file. This information includes the filename currently loaded, the command line used when Leak Check was invoked, the date and time when the user's program began executing, and the date and time of the most recent Leak Check results update.



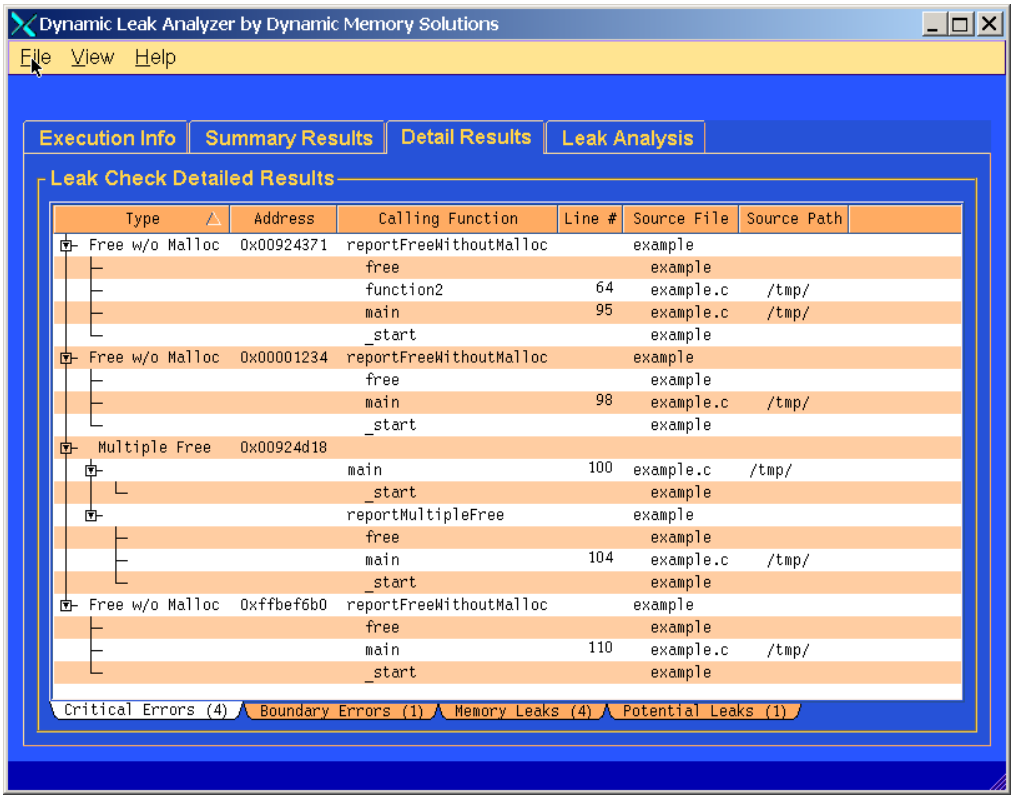
Summary Results

The *Summary Results* tab displays all high-level or summary results for the currently loaded Leak Check binary file. This includes critical errors, boundary errors, leaks and potential leaks.



Detail Results

The *Detail Results* tab contains four sub-tabs that each display a different class of detailed results for the currently loaded Leak Check binary file. These sub-tabs are located at the bottom of the detail table. The four sub-tabs are *Critical Errors*, *Boundary Errors*, *Memory Leaks*, and *Potential Leaks*.



Each sub-tab shows the number of errors reported for that class of detailed results. After a Leak Check binary file is opened, the first sub-tab with errors of that class is automatically selected.

Each sub-tab displays a table showing the details for that class of errors. The columns for each table are specific to the class of errors.

Table columns may be resized by dragging the column separator on the table header.

Table rows may be expanded by clicking the right-facing arrow at the left of the row. Rows may be collapsed by clicking the downward-facing arrow at the left of the row. All rows in the table may also be expanded and collapsed by using the *Expand All* and *Collapse All* commands under the *View* menu.

Table contents may be sorted by clicking on the header of the desired sort column. Clicking the header a second time will reverse the sort order of the column. Only parent (i.e. non-collapsible) rows are sorted.

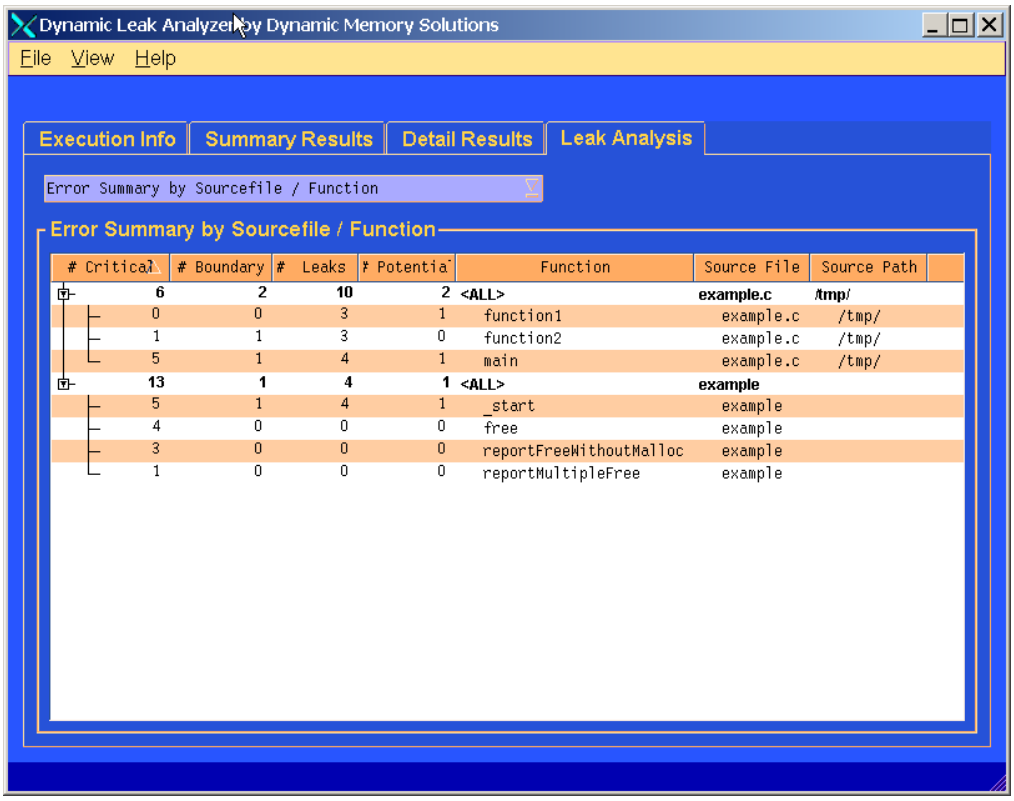
Opening a different Leak Check binary file or using *Refresh* to update the results resets any table formatting changes made by the user.

Leak Analysis

The *Leak Analysis* tab provides a variety of ways to view Leak Check results. These new ways of looking at the results are intended to assist the user in understanding how and where memory allocation errors are being introduced into their applications.

Leak Analysis is expandable and additional capabilities and views are expected to be added with each release. Initially, the *Leak Analysis* tab is a blank canvas, ready to be populated with tables, graphs, and other forms of information display.

A specific view is selected via the drop down list just below the upper window tabs. When a view is selected, data from the currently loaded Leak Check binary file is processed and displayed in the selected view format.



If the view contains a table, table columns may be resized by dragging the column separator on the table header. Table rows may be expanded by clicking the right-facing arrow at the left of the row. Rows may be collapsed by clicking the downward-facing arrow at the left of the row. All rows in the table may also be expanded and collapsed by using the *Expand All* and *Collapse All* commands under the *View* menu. Table contents may be sorted by clicking on the header of the desired sort column. Clicking the header a second time will reverse the sort order of the column. Only parent (i.e. non-collapsible) rows are sorted.

Opening a different Leak Check binary file or using *Refresh* to update the results resets the view chosen by the user.

Limitations of Leak Check

Leak Check can be used to locate memory leaks and other memory errors without recompiling or relinking the application. The information available to Leak Check when reporting errors is limited to the source code information that is included in the executable program and shared libraries.

Known Limitations

1. Leak Check only reports line and system information when objects and/or debug information is available. Source code compiled without the debug option (-g) will not include source line debugging information. Leak Check will detect memory errors, and the function allocating the leaked blocks, but will be unable to report the exact line where the allocation occurred. The DYNPATH option can be used to locate the missing object files if more detailed information is necessary.

2. Leak Check cannot check programs that statically link malloc or free.

3. Heap allocations in main() are reported as leaked if main does not call exit.

FAQ

This section answers many commonly asked questions about Leak Check

- Do I have to recompile or re-link my application to use Leak Check?

  No, recompilation or re-link is not required. The application must be compiled with the debug flag, -g, to get full information. If the executable is fully stripped, little information will be available. If a shared library is stripped, function level information will still be available. If an application is a mix of stripped and not stripped libraries, Leak Check will provide as much information as is available.

- Which platforms does Leak Check run on?

  - Solaris 6, 7, 8, 9, 10 UltraSparc 32 bit / 64 bit
  - Solaris 10 x86 32 bit / 64 bit
  - Linux x86 32 bit / 64 bit (including AMD Opteron)
  - Linux Itanium 64 bit
  - HP-UX 11i Itanium 64 bit

- The output does not contain line number information for some functions, what should I do?

  To get line number information, the application must contain debug (-g) information and must not be stripped. Some parts of the application can be stripped while other parts contain debug information. Leak Check will always give as much information as is available.

- How can I use Leak Check with a script that starts my application?

  The easiest way is to just leak check every process that is started by the script by exporting the LD_PRELOAD variable into the environment. An example can be seen in the $DYNROOT/env/leak32.env file. If this file is sourced then any (32-bit) processes started from that terminal will be checked for leaks. The output will appear in the $DYNOUTPUT directory or in /tmp.

  As an alternative, the master script can be modified to kick off Leak Check on the target application. The $DYNROOT/bin/dynleak file is a script and a useful example.

- My application crashes when I use Leak Check, what should I do?

  Try the **DYNNOFREE** option. Sometimes a bug in the application will cause the program to crash. By using each block of memory only once, the program may avoid the crash. If you are still having problems, please contact our technical support staff at support@dynamc-memory.com. We will work with you to determine the problem.

- I am having problems with my Solaris 64-bit program compiled with gcc. It works fine when I do not use the Leak Check, but I get an ELF class error when running with the DLC. What can I do?

  Include /usr/local/lib/sparcv9 in your LD_LIBRARY_PATH before /usr/local/lib.

- When I run my Solaris application with Leak Check, I am told that libdemangle.so.1 is missing. Why?

  Leak Check has a requirement that the SUN SUNWlibC package is installed. Please have this package installed and try the tool again.

- Is Leak Check thread-safe?

  Yes, Leak Check is thread-safe on all platforms and word sizes.

You're gonna need stronger debugging software

Document Number DLC20-L-021-1

www.dynamic-memory.com