

# Malloc Debug Environment Variables Release Notes: Debugging environment variables for the standard malloc package

The standard malloc package found in System.framework provides extra debugging features which can be turned on and off at runtime. These options allow you to gather information about which functions are allocating memory and encourage memory problems to surface. These flags are intended to help you track down memory smashers, heap corruption, references to freed memory, and buffer overruns.

Other memory allocation debugging libraries in OS X include libgmalloc and libMallocDebug. For information about libgmalloc, run 'man libgmalloc' to read the libgmalloc(3) man page. libgmalloc can also be used when debugging in Xcode by using the "Enable Guard Malloc" menu item in the Debug menu. To use libMallocDebug you can run your application from the MallocDebug application.

The following environment variables control the extra debugging features in the standard malloc. These are also documented on the malloc man page.

- MallocStackLogging causes malloc to remember the caller requesting each allocation.
- MallocStackLoggingNoCompact causes malloc to remember the caller requesting each allocation, and remembers it after the allocation is freed.
- MallocPreScribble detects reads of uninitialized memory by setting its contents to 0xAA when allocated.
- MallocScribble detects writes to a freed block by setting its contents to 0x55 when deallocated.
- MallocGuardEdges add guard pages before and after large allocations
- MallocDoNotProtectPrelude: disables the guard page before block
- MallocDoNotProtectPostlude disables the guard page after block
- MallocCheckHeapStart checks the heap's internal data structures at intervals.
- MallocCheckHeapEach checks the heap's internal data structures at intervals.
- MallocCheckHeapSleep causes sleep() to be called if a heap corruption is detected.
- MallocCheckHeapAbort causes abort() to be called if a heap corruption is detected.
- MallocBadFreeAbort causes abort() to be called if an illegal free() call is made.

The options are enabled by creating an environment variable with the name of the option. These can also be set when debugging with Xcode, by choosing an executable in the Executables group in the Groups & Files outline, doing Get Info on the Executable, going to the Arguments tab, and entering the environment variables in the table in the bottom of the Get Info window.

## Contents:

- Description of each option
- Turning on the options
- Options needed by performance tools
- Hints at debugging with the malloc options

## Description of each option

### MallocStackLogging and MallocStackLoggingNoCompact

MallocStackLogging tells the malloc library to memorize who allocated the memory. For each allocation, MallocDebug remembers the stack backtrace -- the chain of functions that were called in order to call malloc. This information is used by tools such as leaks and heap to identify the purpose of a specific allocation.

### MallocStackLoggingNoCompact

MallocStackLoggingNoCompact does the same task as MallocStackLogging, but remembers stack traces about allocations that have been freed and no longer exist. This flag is necessary to use the malloc\_history command, which uses the feature to identify all the blocks allocated at a specific address during the lifetime of the program.

### MallocPreScribble

If a block of memory is allocated and a function tries to read a pointer from that memory before the pointer has been initialized, the program could occasionally crash when dereferencing that pointer. If MallocPreScribble is set, the malloc library writes newly allocated memory with the value 0xAA. Reading an uninitialized pointer from that memory will cause the program to reference the memory at 0xAAAAAAAA, which is usually unallocated and will cause an immediate crash if the pointer is dereferenced for reading or writing.

### MallocScribble

If a pointer is kept to a block of memory after the memory has been freed, and the block is reallocated, a function using the out-of-date pointer could overwrite the new contents of the buffer, confusing the program and causing occasional crashes. MallocScribble helps you find such problems by overwriting the contents of freed memory, ensuring that if the application tries to read the memory after it has been freed, but before it has been reallocated, that the reader will find garbage values. The malloc library overwrites freed memory with the value 0x55 if MallocScribble is set. Dereferencing a pointer in cleared freed memory will cause the program to reference the memory at 0x55555555, which usually is unallocated and will cause an immediate crash if the pointer is dereferenced for reading or writing.

### MallocGuardEdges, MallocDoNotProtectPrelude, MallocDoNotProtectPostlude

In order to catch buffer overruns, malloc can protect memory before and after the buffer so that reads or writes will cause a bad memory access and crash the program. With the MallocGuardEdges set, guard pages are placed on each side of large (4096 bytes or more) buffers.

MallocDoNotProtectPrelude and MallocDoNotProtectPostlude remove the guard page on one end of the block. These options have no effect on memory allocations smaller than 4096 bytes.

### MallocCheckHeapStart, MallocCheckHeapEach

These options allow you to check for corruption of malloc's own internal data structures (via a badly-behaved program writing where it shouldn't) by running the heap checking routines at intervals. The MallocCheckHeapStart is set to the number of allocations that should be performed before heap checking is performed, and MallocCheckHeapEach specifies the interval after that point that checking should be performed. As the program runs, it will print a message at each successful check, naming the total number of allocations performed:

```
MallocCheckHeap: PASSED check at 37800th operation
```

### MallocCheckHeapSleep, MallocCheckHeapAbort

You can set MallocCheckHeapSleep to cause a program to sleep when a corruption occurs, you should be able to identify the number of allocations that have probably been performed, and can then adjust MallocCheckHeapStart and MallocCheckHeapEach to better refine the guess about where the problem occurs. When you are close to the source of the problem, you can set MallocCheckHeapAbort to break into the debugger when the corruption has occurred. Or, you can set MallocCheckHeapSleep to allow attaching the debugger.

## Turning on the options

The above options are turned on and off using environment variables. The variables can be set in several ways. First, the environment variable can be set in the shell immediately before executing the program to be tested. Variables that are on or off are set by defining or undefining the symbol. Variables such as `MallocCheckHeapStart` and `MallocCheckHeapEach` are set to the desired value of checking. Variables can be turned off with `unsetenv`.

```
% setenv MallocStackLogging any_value
% setenv MallocCheckHeapStart 1000
% setenv MallocCheckHeapEach 100
% MyApp
```

Second, you could set the environment variables in your shell's startup file so that they're always used. Finally, if you run the program in `gdb`, you can set environment variables inside `gdb` by using the command `"set env"`:

```
% gdb appname
<gdb> set env MallocStackLogging 1
<gdb> run
```

## Options needed by performance tools

Some of the performance tools require these options to be set in order to gather their data. `malloc_history`, a command line tool, can identify the allocation site of specific blocks if the `MallocStackLogging` flag is set, and can describe the blocks previously allocated at an address if the `MallocStackLoggingNoCompact` environment variable is set. The `leaks` command line tool will name the allocation site of a leaked buffer if `MallocStackLogging` is set. See the man page for `leaks` and `malloc history` for more details.

Man pages are viewed by typing `"man malloc_history"` from the command line.

## Hints at debugging with the malloc options

### Tracking down memory smashers

Try to run your program with `MallocPreScribble` and `MallocScribble` often. Memory smasher bugs -- the bugs it catches -- are the sort that are subtle and difficult to track down, so any help you can get to identify that such bugs exist can make a great difference in your code's reliability. Also set `MallocStackLogging` and `MallocStackLoggingNoCompact`. When you find freed memory being cleared, use the `malloc_history` command to see who had allocated memory at that location in the past, and search your code for cases where a pointer to such an allocation might be kept around even after the block has been freed.

### Where am I performing that double free?

If you see a message (such as the warning that an address that's not a malloc-allocated buffer has been passed to free), you can track down the location of the free by running the program in the debugger with `MallocBadFreeAbort` set. If the program breaks into the debugger when a bad free is done, you can examine the stack backtrace to figure out which free is receiving the bogus value.

You can also put a breakpoint on `malloc_printf` to break when the malloc system prints warning messages. `malloc` will print such warnings in cases of invalid entries, such as a requested size exceeds the maximum block size allowed, or a pointer passed to free does not reference a currently allocated block.

## Tracking down heap corruption

To track down possible heap corruption, you can set the heap validation environment variables, `MallocCheckHeapStart`, `MallocCheckHeapEach`, and `MallocCheckHeapAbort` (or `MallocCheckHeapSleep`). Set the first to the number of mallocs that should happen before it validates the correctness of the heap, and the second to the interval between checks. If you've got a case where you're corrupting the heap and can easily reproduce the problem at the same point in the program every time, you can use these two settings to narrow the problem down to a specific malloc. Set the `MallocCheckHeapStart` to (say) 1000, and `MallocCheckHeapEach` to 1000. Watch malloc print out the number of allocations performed as it goes along, and figure out about when the heap starts to appear invalid. Stop the program, then either set `Start` and `Each` to narrow down the place where the problem occurs, and rerun the program. With this, you should have some chance to figure out where the corruption's occurring.