Heap Analysis: Making Memory Errors a Thing of the Past#

Introduction#

If you develop a program that dynamically allocates memory, you're also responsible for tracking any memory that you allocate whenever a task is performed, and for releasing that memory when it's no longer required. If you fail to track the memory correctly you may introduce "memory leaks," or unintentionally write to an area outside of the memory space.

Conventional debugging techniques usually prove to be ineffective for locating the source of corruption or leak because memory-related errors typically manifest themselves in an unrelated part of the program. Tracking down an error in a multithreaded environment becomes even more complicated because the threads all share the same memory address space.

Dynamic memory management#

In a program, you'll dynamically request memory buffers or blocks of a particular size from the runtime environment using malloc(), realloc(), or calloc(), and then you'll release them back to the runtime environment when they're no longer required using free().

The memory allocator ensures that your requests are satisfied by managing a region of the program's memory area known as the heap. In this heap, it tracks all of the information -- such as the size of the original block -- about the blocks and heap buffers that it has allocated to your program, in order that it can make the memory available to you during subsequent allocation requests. When a block is released, it places it on a list of available blocks called a free list. It usually keeps the information about a block in the header that precedes the block itself in memory.

The runtime environment grows the size of the heap when it no longer has enough memory available to satisfy allocation requests, and it returns memory from the heap to the system when the program releases memory.

Heap corruption#

Heap corruption occurs when a program damages the allocator's view of the heap. The outcome can be relatively benign and cause a memory leak (where some memory isn't returned to the heap and is inaccessible to the program afterwards), or it may be fatal and cause a memory fault, usually within the allocator itself. A memory fault typically occurs within the allocator when it manipulates one or more of its free lists after the heap has been corrupted.

It's especially difficult to identify the source of corruption when the source of the fault is located in another part of the code base. This is likely to happen if the fault occurs when:

- a program attempts to free memory
- a program attempts to allocate memory after it's been freed
- the heap is corrupted long before the release of a block of memory
- the fault occurs on a subsequent block of memory
- contiguous memory blocks are used
- your program is multithreaded
- the memory allocation strategy changes

Contiguous memory blocks#

When contiguous blocks are used, a program that writes outside of the bounds can corrupt the allocator's information about the block of memory it's using, as well as, the allocator's view of the heap. The view may

include a block of memory that's before or after the block being used, and it may or may not be allocated. In this case, a fault in the allocator will likely occur during an unrelated allocation or release attempt.

Multithreaded programs#

Multithreaded execution may cause a fault to occur in a different thread from the thread that actually corrupted the heap, because threads interleave requests to allocate or release memory.

When the source of corruption is located in another part of the code base, conventional debugging techniques usually prove to be ineffective. Conventional debugging typically applies breakpoints -- such as stopping the program from executing -- to narrow down the offending section of code. While this may be effective for single-threaded programs, it's often unyielding for multithreaded execution because the fault may occur at an unpredictable time and the act of debugging the program may influence the appearance of the fault by altering the way that thread execution occurs. Even when the source of the error has been narrowed down, there may be a substantial amount of manipulation performed on the block before it's released, particularly for long-lived heap buffers.

Allocation strategy#

A program that works in a particular memory allocation strategy may abort when the allocation strategy is changed in a minor way. A good example of this would be a memory overrun condition (for more information see "Overrun and underrun errors," below) where the allocator is free to return blocks that are larger than requested in order to satisfy allocation requests. Under this circumstance, the program may behave normally in the presence of overrun conditions. But a simple change, such as changing the size of the block requested, may result in the allocation of a block of the exact size requested, resulting in a fatal error for the offending program.

Fatal errors may also occur if the allocator is configured slightly differently, or if the allocator policy is changed in a subsequent release of the runtime library. This makes it all the more important to detect errors early in the life cycle of an application, even if it doesn't exhibit fatal errors in the testing phase.

Common sources#

Some of the most common sources of heap corruption include:

- a memory assignment that corrupts the header of an allocated block
- an incorrect argument that's passed to a memory allocation function
- an allocator that made certain assumptions in order to avoid keeping
- additional memory to validate information, or to avoid costly runtime checking
- invalid information that's passed in a request, such as to free().
- overrun and underrun errors
- releasing memory
- using uninitialized or stale pointers

Even the most robust allocator can occasionally fall prey to the above problems.

Let's take a look at the last three bullets in more detail:

Overrun and underrun errors#

Overrun and underrun errors occur when your program writes outside of the bounds of the allocated block. They're one of the most difficult type of heap corruption to track down, and usually the most fatal to program execution.

Overrun errors occur when the program writes past the end of the allocated block. Frequently this causes corruption in the next contiguous block in the heap, whether or not it's allocated. When this occurs, the behavior that's observed varies depending on whether that block is allocated or free, and whether it's associated with a part of the program related to the source of the error. When a neighboring block that's allocated becomes corrupted, the corruption is usually apparent when that block is released elsewhere in the program. When an unallocated block becomes corrupted, a fatal error will usually result during a subsequent allocation request. Although this may well be the next allocation request, it's actually dependent on a complex set of conditions that could result in a fault at a much later point in time, in a completely unrelated section of the program, especially when small blocks of memory are involved.

Underrun errors occur when the program writes before the start of the allocated block. Often they corrupt the header of the block itself, and sometimes, the preceding block in memory. Underrun errors usually result in a fault that occurs when the program attempts to release a corrupted block.

Releasing memory#

Requests to release memory requires your program to track the pointer for the allocated block and pass it to the free() function. If the pointer is stale, or if it doesn't point to the exact start of the allocated block, it may result in heap corruption.

A pointer is stale when it refers to a block of memory that's already been released. A duplicate request to free() involves passing free() a stale pointer -- there's no way to know whether this pointer refers to unallocated memory, or to memory that's been used to satisfy an allocation request in another part of the program.

Passing a stale pointer to free() may result in a fault in the allocator, or worse, it may release a block that's been used to satisfy another allocation request. If this happens, the code making the allocation request may compete with another section of code that subsequently allocated the same region of heap, resulting in corrupted data for one or both. The most effective way to avoid this error is to NULL out pointers when the block is released, but this is uncommon, and difficult to do when pointers are aliased in any way.

A second common source of errors is to attempt to release an interior pointer (i.e. one that's somewhere inside the allocated block rather than at the beginning). This isn't a legal operation, but it may occur when the pointer has been used in conjunction with pointer arithmetic. The result of providing an interior pointer is highly dependent on the allocator and is largely unpredictable, but it frequently results in a fault in the free() call.

A more rare source of errors is to pass an uninitialized pointer to free(). If the uninitialized pointer is an automatic (stack) variable, it may point to a heap buffer, causing the types of coherency problems described for duplicate free() requests above. If the pointer contains some other nonNULL value, it may cause a fault in the allocator.

Using uninitialized or stale pointers#

If you use uninitialized or stale pointers, you might corrupt the data in a heap buffer that's allocated to another part of the program, or see memory overrun or underrun errors.

Detecting and reporting errors#

The primary goal for detecting heap corruption problems is to correctly identify the source of the error, rather than getting a fault in the allocator at some later point in time.

A first step to achieving this goal is to create an allocator that's able to determine whether the heap was corrupted on every entry into the allocator, whether it's for an allocation request or for a release request. For example, on a release request, the allocator should be capable of determining whether:

- the pointer given to it is valid
- the associated block's header is corrupt
- either of the neighboring blocks are corrupt

To achieve this goal, we'll use a replacement library for the allocator that can keep additional block information in the header of every heap buffer. This library may be used during the testing of the application to help isolate any heap corruption problems. When a source of heap corruption is detected by this allocator, it can print an error message indicating:

- the point at which the error was detected
- the program location that made the request
- information about the heap buffer that contained the problem

The library technique can be refined to also detect some of the sources of errors that may still elude detection, such as memory overrun or underrun errors, that occur before the corruption is detected by the allocator. This may be done when the standard libraries are the vehicle for the heap corruption, such as an errant call to memcpy(), for example. In this case, the standard memory manipulation functions and string functions can be replaced with versions that make use of the information in the debugging allocator library to determine if their arguments reside in the heap, and whether they would cause the bounds of the heap buffer to be exceeded. Under these conditions, the function can then call the error reporting functions to provide information about the source of the error.

Using the malloc debug library#

The malloc debug library provides the capabilities described in the above section. It's available when you link to either the normal memory allocator library, or to the debug library:

To access:	Link using this option:
Nondebug library	-lmalloc
Debug library	-lmalloc_g

If you use the debug library, you must also include:

`/usr/lib/malloc_g`

as the first entry of your $LD_LIBRARY_PATH environment variable before running your application.

Another option to use the debug malloc library is to use the LD_PRELOAD capability to the dynamic loader. The LD_PRELOAD environment variable allows you to specify libraries that will loaded prior to any other library in the system. In this case, the LD_PRELOAD variable should be set to point to the location of the debug malloc library (or the non-debug one as the case may be), by saying

`LD_PRELOAD=/usr/lib/malloc_g/libmalloc.so.2` or `LD_PRELOAD=/usr/lib/libmalloc.so.2`

In this chapter all references to the malloc library refer to the debug version, unless otherwise specified.

Both versions of the library share the same internal shared object name, so it's actually possible to link against the nondebug library and test using the debug library when you run your application. To do this, you must change the $LD_LIBRARY_PATH as indicated above.

The nondebug library doesn't perform heap checking; it provides the same memory allocator as the system library.

By default, the malloc library provides a minimal level of checking. When an allocation or release request is performed, the library checks only the immediate block under consideration and its neighbors looking for sources of heap corruption.

Additional checking and more informative error reporting can be done by using additional calls provided by the malloc library. The mallopt() function provides control over the types of checking performed by the library. There are also debug versions of each of the allocation and release routines that can be used to provide both file and line information during error reporting. In addition to reporting the file and line information about the caller when an error is detected, the error reporting mechanism prints out the file and line information that was associated with the allocation of the offending heap buffer.

To control the use of the malloc library and obtain the correct prototypes for all the entry points into it, it's necessary to include a different header file for the library. This header file is included in <malloc_g/malloc.h>. If you want to use any of the functions defined in this header file, other than mallopt(), make sure that you link your application with the debug library. If you forget, you'll get undefined references during the link.

The recommended practice for using the library is to always make use of the library for debug variants in builds. In this case, the macro used to identify the debug variant in C code should trigger the inclusion of the <malloc_g/malloc.h> header file, and the malloc debug library option should always be added to the link command. In addition, you may want to follow the practice of always adding an exit handler that provides a dump of leaked memory, and initialization code that turns on a reasonable level of checking for the debug variant of the program.

The malloc library achieves what it needs to do by keeping additional information in the header of each heap buffer. The header information includes additional storage for keeping doubly-linked lists of all allocated blocks, file, line and other debug information, flags and a CRC of the header. The allocation policies and configuration are identical to the normal system memory allocation routines except for the additional internal overhead imposed by the malloc library. This allows the malloc library to perform checks without altering the size of blocks requested by the program. Such manipulation could result in an alteration of the behavior of the program with respect to the allocator, yielding different results when linked against the malloc library.

All allocated blocks are integrated into a number of allocation chains associated with allocated regions of memory kept by the allocator in arenas or blocks. The malloc library has intimate knowledge about the internal structures of the allocator, allowing it to use short-cuts to find the correct heap buffer associated with any pointer, resorting to a lookup on the appropriate allocation chain only when necessary. This minimizes the performance penalty associated with validating pointers, but it's still significant.

The time and space overheads imposed by the malloc library are too great to make it suitable for use as a production library, but are manageable enough to allow them to be used during the test phase of development and during program maintenance.

What's checked?#

As indicated above, the malloc library provides a minimal level of checking by default. This includes a check of the integrity of the allocation chain at the point of the local heap buffer on every allocation request. In addition, the flags and CRC of the header are checked for integrity. When the library can locate the neighboring heap buffers, it also checks their integrity. There are also checks specific to each type of allocation request that are done. Call-specific checks are described according to the type of call below.

Additional checks can be turned on using the mallopt() call. For more information on the types of checking, and the sources of heap corruption that can be detected, see "Controlling the level of checking," below.

Allocating memory#

When a heap buffer is allocated using any of the heap allocation routines, the heap buffer is added to the allocation chain for the arena or block within the heap that the heap buffer was allocated from. At this time, any problems detected in the allocation chain for the arena or block are reported. After successfully inserting the allocated buffer in the allocation chain, the previous and next buffers in the chain are also checked for consistency.

Reallocating memory#

When an attempt is made to resize a buffer through a call to the realloc() function, the pointer is checked for validity if it's a nonNULL value. If it's valid, the header of the heap buffer is checked for consistency. If the buffer is large enough to satisfy the request, the buffer header is modified and the call returns. If a new buffer is required to satisfy the request, memory allocation is performed to obtain a new buffer large enough to satisfy the request with the same consistency checks being applied as in the case of memory allocation described above. The original buffer is then released.

If fill-area boundary checking is enabled (described in the "Manual Checking" section) the guard code checks are also performed on the allocated buffer before it's actually resized. If a new buffer is used, the guard code checks are done just before releasing the old buffer.

Releasing memory#

This includes, but isn't limited to, checking to ensure that the pointer provided to a free() request is correct and points to an allocated heap buffer. Guard code checks may also be performed on release operations to allow fill-area boundary checking.

Controlling the level of checking#

The mallopt() function call allows extra checks to be enabled within the library. The call to mallopt requires that the application is aware that the additional checks are programmatically enabled. The other way to enable the various levels of checking is to use environment variables for each of the mallopt options. Using environment variables allows the user to specify options that will be enabled from the time the program runs, as opposed to only when the code that triggers these options to be enabled (i.e. the mallopt call) is reached. For certain programs that perform a lot of allocations before main(), setting options using mallopt calls from main() or after that may be too late. In such cases it is better to use environment variables.

int mallopt (int cmd, int value);

cmd
  An integer that indicates the parameter (or option) to be
  affected by the call.
  Options used to enable additional checks in the library include:

MALLOC_CKACCESS
  Turn on (or off) boundary checking for memory and string
  operations.
  Environment variable: MALLOC_CKACCESS
MALLOC_FILLAREA
  Turn on (or off) fill-area boundary checking.
  Environment variable: MALLOC_FILLAREA
MALLOC_CKCHAIN
  Enable (or disable) full-chain checking.
  For each of the above options, an integer argument value of one

indicates that the given type of checking should be enabled from
that point onward.
Environment variable: MALLOC_CKCHAIN

value
Indicate whether checking is to be performed. An integer value
that can hold any legal value for malloc options or parameters.

Description of optional checks#

`MALLOC_CKACCESS`

Turn on (or off) boundary checking for memory and string operations. This helps to detect buffer overruns and underruns that are a result of memory or string operations. When on, each pointer operand to a memory or string operation is checked to see if it's a heap buffer. If it is, the size of the heap buffer is checked and the information is used to ensure that no assignments are made beyond the bounds of the heap buffer. If an attempt is made that would assign past the buffer boundary, a diagnostic warning message is printed.

Here's how you can use this option to find an overrun error:

```
...
char *p;
int opt;
opt = 1;
mallopt(MALLOC_CKACCESS, opt);
p = malloc(strlen("hello"));
strcpy(p, "hello, there!");  /* a warning is generated
                                here */
```

...The following illustrates how access checking can trap a reference through a stale pointer:

```
...
char *p;
int opt;
opt = 1;
mallopt(MALLOC_CKACCESS, opt);
p = malloc(30);
free(p);
strcpy(p, "hello, there!");
```

`MALLOC_FILLAREA`

Turn on (or off) fill-area boundary checking. Fill-area boundary checking validates that the program hasn't overrun the user-requested size of a heap buffer. It does this by applying a guard code check when the buffer is released or when it's resized. The guard code check works by filling any excess space available at the end of the heap buffer with a pattern of bytes. When the buffer is released or resized, the trailing portion is checked to see if the pattern is still present. If not, a diagnostic warning message is printed.

The effect of turning on fill-area boundary checking is a little different than enabling other checks. The checking is performed only on memory buffers allocated after the point in time at which the check was enabled. Memory buffers allocated before the change won't have the checking performed.

Here's how you can catch an overrun with the fill-area boundary checking option:

```
...
int *foo, *p, i, opt;
opt = 1;
mallopt(MALLOC_FILLAREA, opt);
```

```
foo = (int *)malloc(10*4);
for (p = foo, i = 12; i > 0; p++, i--)
   *p = 89;
free(foo);  /* a warning is generated here */
```

`MALLOC_CKCHAIN`

Enable (or disable) full-chain checking. This option is expensive and should be considered as a last resort when some code is badly corrupting the heap and otherwise escapes the detection of boundary checking or fill-area boundary checking. This can occur under a number of circumstances, particularly when they're related to direct pointer assignments. In this case, the fault may occur before a check such as fill area boundary checking can be applied. There are also circumstances in which both fill-area boundary checking and the normal attempts to check the headers of neighboring buffers fails to detect the source of the problem. This may happen if the buffer that's overrun is the first or last buffer associated with a block or arena. It may also happen when the allocator chooses to satisfy some requests, particularly those for large buffers, with a buffer that exactly fits the program's requested size.

Full-chain checking traverses the entire set of allocation chains for all arenas and blocks in the heap every time a memory operation (including allocation requests) is performed. This allows the developer to narrow down the search for a source of corruption to the nearest memory operation.

Forcing verification#

You can force a full allocation chain check at certain points while your program is executing, without turning on chain checking. Specify the following option for cmd:

`MALLOC_VERIFY`

Perform a chain check immediately. If an error is found, perform error handling.

Specifying an error handler#

Typically, when the library detects an error, a diagnostic message is printed and the program continues executing. In cases where the allocation chains or another crucial part of the allocator's view is hopelessly corrupted, an error message is printed and the program is aborted (via abort()).

You can override this default behavior by specifying a handler that determines what is done when a warning or a fatal condition is detected.

cmd
   Specify the error handler to use.
MALLOC_FATAL
   Specify the malloc fatal handler.
   Environment variable: MALLOC_FATAL
MALLOC_WARN
   Specify the malloc warning handler handler.
   Environment variable: MALLOC_WARN

value
An integer value that indicates which one of the standard
handlers provided by the library.

   M_HANDLE_ABORT
      Terminate execution with a call to abort().
   M_HANDLE_EXIT
      Exit immediately.

M_HANDLE_IGNORE
Ignore the error and continue.
M_HANDLE_CORE
Cause the program to dump a core file
M_HANDLE_SIGNAL
Stop the program when this error occurs, by sending itself a stop signal. This allows one to attach to this process using a debugger. The program is stopped inside the error handler function, and a backtrace from there should show one the exact location of the error.

If using environment variables to specify options to the malloc library for either MALLOC_FATAL or MALLOC_WARN, the values passed for each of the possibilities needs to be explcitly specified (instead of the symbolic name for the error handling level)

Handler	Value
M_HANDLE_IGNORE	0
M_HANDLE_ABORT	1
M_HANDLE_EXIT	2
M_HANDLE_CORE	3
M_HANDLE_SIGNAL	4

for this release. These values are also defined in `/usr/include/malloc_g/malloc-lib.h`

Any of these handlers can be ORed with the value, MALLOC_DUMP, to cause a complete dump of the heap before the handler takes action.

Here's how you can cause a memory overrun error to abort your program:

```
...
int *foo, *p, i;
int opt;
opt = 1;
mallopt(MALLOC_FILLAREA,  opt);
foo = (int *)malloc(10*4);
for (p = foo, i = 12; i > 0; p++, i--)
   *p = 89;
opt = M_HANDLE_ABORT;
mallopt(MALLOC_WARN, opt);
free(foo); /* a fatal error is generated here */
```

Other environment variables#

`MALLOC_INITVERBOSE`

Enable some initial verbose output regarding other variables that are enabled.

`MALLOC_BTDEPTH`

this variable set the depth of the backtrace, on CPUs that support deeper backtrace levels. Currently the builtin-return-address feature of GCC is used to implement deeper backtraces for the debug malloc library. This variable controls the depth of the backtrace for allocations (i.e. where the allocation occured)

`MALLOC_TRACEBT`

this variable set the depth of the backtrace, on CPUs that support deeper backtrace levels. Currently the builtin-return-address feature of GCC is used to implement deeper backtraces for the debug malloc library. This variable controls the depth of the backtrace for errors and warning.

`MALLOC_DUMP_LEAKS`

this variable triggers leak detection on exit of the program. The output of the leak detection is sent to a file that is specified by a name that is the argument to this variable.

`MALLOC_TRACE`

this enables tracing of all calls to malloc/free/calloc/realloc etc. A trace of the various calls is made available in a file specified by the argument to this variable name.

`MALLOC_CKACCESS_LEVEL`

this variable allows one to specify the level of checking performed by the MALLOC_CKACCESS option. By default a basic level of checking is performed. By increasing the LEVEL of checking, additional things that could be errors are also flagged. For example a call to memset with a length of zero is normally safe, since no data is actually moved, but it the arguments point to illegal locations (memory references that are invalid), this normally suggests a case where there is a problem potentially lurking inside the code. By increasing the level of checking, these kinds of errors are also flagged.

Caveats#

The debug malloc library when enabled with a lot of checking, uses more stack space (i.e. calls more functions, uses more local variables etc) than the regular libc allocator. This implies that programs that explicitly set the stack size to something smaller than the default may encounter problems like running out of stack space, causing the program to crash. This can be fixed by increasing the stack space allocated to the threads in question.

MALLOC_FILLAREA is used to do fill area checking. If fill area checking is not enabled, certain types of errors cannot be detected. For example errors where an application accesses beyond the end of a block. but the real block allocated by the allocator is larger than what was requested, the allocator will not flag an error unless MALLOC_FILLAREA is enabled. By default this is _not_ enabled.

MALLOC_CKACCESS is used to validate accesses to the str* and mem* family of functions. If this variable is not enabled, such accesses will not be checked, and errors not reported. By default this is _not_ enabled.

MALLOC_CKCHAIN is used to perform entensive heap checking on every allocation. If this is enabled, it can make allocations much slower. Also since full heap checking is performed on every allocation, an error anywhere in the heap could be reported upon entry into the allocator for any operation. For example a call to free(x) would check block x, and also the complete heap for errors before completing the operation to free block x. So any error in the heap will be reported in the context of free-ing block x, even if the error itself is not specifically related to operation of free-ing block x.

When the debug library reports errors, it does not always exit immediately, instead it continues on to perform the operation that causes the error, and actually can corrupt the heap (since that operation that raises the warning is actually an illegal operation). This behaviour can be controlled by using the "MALLOC_WARN" and MALLOC_FATAL handler described earlier. If specific handlers are not provided, the heap will be corrupted, and potentially other errors will be reported later that are a result of this first error. The best solution is to focus on the first error, fix it before moving onto other errors, since they may actually be just a result of the first one itself, or to use error handling capabilities that will result in the program not making progress beyond the first error. (Look at description of MALLOC_CKCHAIN for more information of how these sorts of errors may end up getting reported).

Although the debug malloc library allocates blocks to the user using the same algorithms as the standard allocator, since it uses additional storage to maintain block information, and to be able to perform sanity checks the storage requirements when using the debug allocator are higher than when using the standard allocator. This means the layout of blocks in memory when using the debug allocator may be slightly different than when using the standard allocator.

Another caveat must be mentioned here. The use of certain optimization options such as -O1, -O2 or -O3 do not allow the malloc debug library to work correctly. The problem occurs due to the fact that, during compilation and linking, the gcc call builtin functions instead of calling intended functions such as strcpy() or strcmp(). You should use -fno-builtin option to circumvent this problem.