Red Zone Memory Allocator under Mach 3.0

Tero Kivinen <kivinen@hut.fi>,
Helsinki University of Technology

Thursday, April 9, 1992

Abstract

This article describes an implementation of Redzal, a red zone memory allocator using virtual memory features of mach 3.0. This malloc replacement is neither efficient nor portable, and its current implementation uses huge amounts of memory, but it serves excellently its primary purpose: it can detect bugs.

1.0 Why is a red zone memory allocator needed?

1.1 Overwriting data

One of the most common errors in C-programs is illegal use of allocated memory. For example, a program allocates a buffer for an input stream and then reads data to it. If the data does not fit to the allocated area and the program does not check this, the read can overwrite important data located after the buffer. When this data is later used, random errors can occur at places that are not related to the place where the actual error occurred. Had the error been detected right away, fixing it would have taken only a few seconds - now it can take hours. To help debugging this, we need a good debugging malloc.

1.2 Reading out of bounds data

Older debugging malloc packages check after every malloc operation that nobody has overwritten data between allocated areas. This will detect the errors described above, but it will not detect illegal reads outside the allocated area.

The following example was found in the GNU C-compiler (gcc) by this malloc implementation. When compiling itself gcc generates from a machine description file several C-source files, which form the machine depended part of the compiler. One of these generators reads a line from the machine description file and checks if the last character in line is '*'. Unfortunately, it didn't check if line could be empty, so when it did "`if (line [ strlen( line ) - 1 ] == '*')`" it tested a character in front of the allocated area. This character could have been '*' and so this generator could have generated erroneous code, later causing gcc to generate invalid code. Finding this kind of bugs without a red zone memory allocator would be very difficult.

1.3 Other errors

Other types of malloc-related errors are memory leaks, accessing uninitialised, freed or reallocated data. This implementation will detect all accesses to freed or reallocated data, and offers some help to find memory leaks.

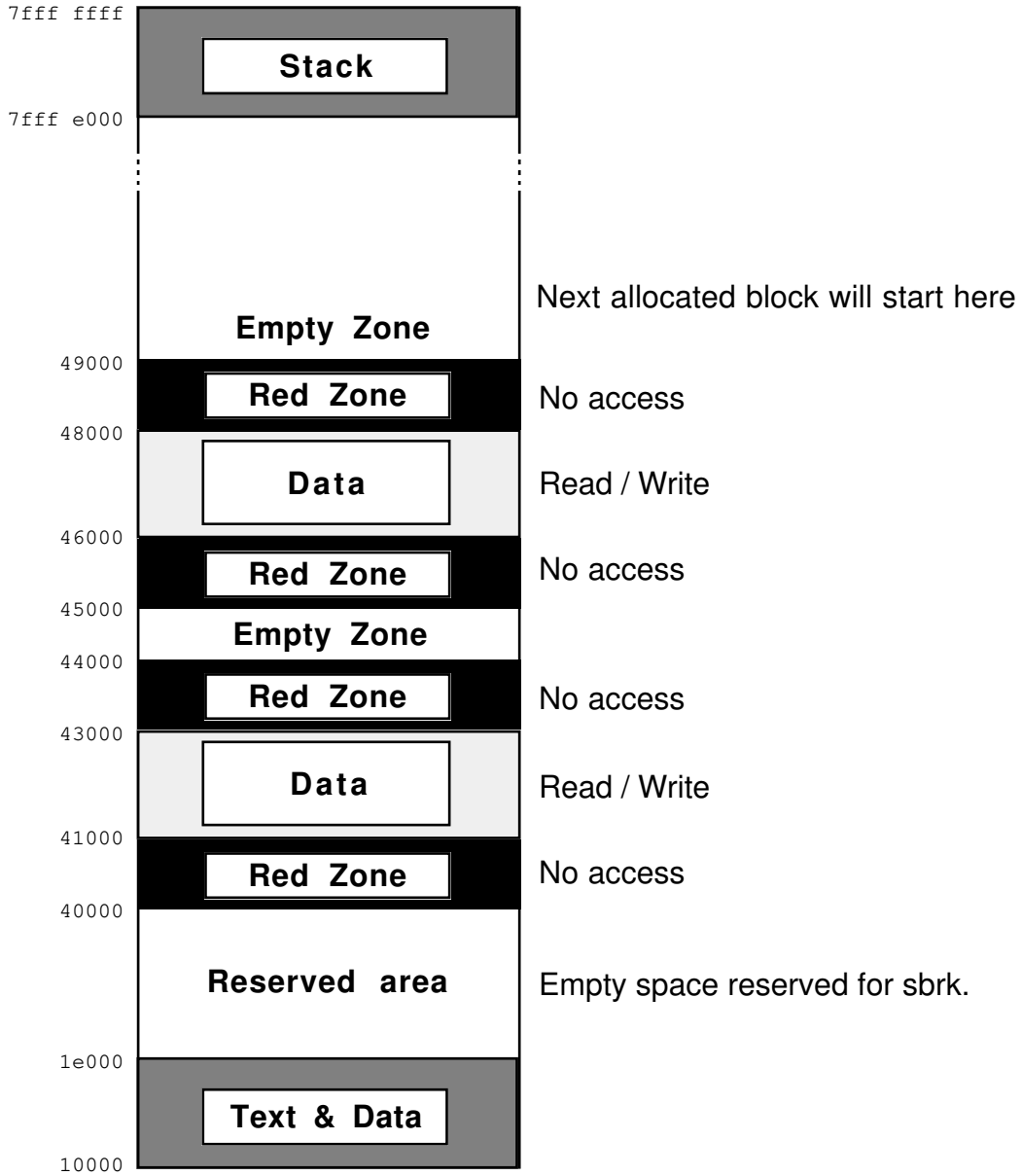
2.0 What is a red zone memory allocator?

A red zone memory allocator (RZMA) creates an area called the red zone around legal data, and detects reads or writes to it. Because allocation routines are often used to allocate arrays of data, the size of the red zone should be at least the size of one element of the allocated array. In this way the RZMA will detect common "off by one" errors when the program tries to accesses elements beyond or before the allocated array. In this implementation, the size of the red zone is at least one page (4 kB) and its can be configured by the user when the program is started.

When the program accesses memory outside the allocated data, the allocator will need a method to find the block the program tried to access. This type of illegal access could also have been a random pointer dereference and no allocated block was involved. This problem is solved in this implementation by creating an empty zone between allocated blocks (which are surrounded by the red zone). So if the red zone is accessed the RZMA displays the nearest data block, and if the empty zone is accessed then it

was probably just a random access, so no block is displayed. By creating larger empty zones the RZMA can identify more random accesses, but it will waste much more virtual address space.

In case of large programs (gcc, zsh, etc.) it will run out of 32-bit virtual memory quite soon, if larger empty zone or red zone is used. Every malloc will need virtual address space of 2 * red zone size + real data size (rounded up to page size) + empty zone size. This means that the allocator can do about 125 000 mallocs when using 4 kB zones, but when using 64 kB zones there can be only 10 000 allocated blocks, assuming normal 32 bit address space. Zsh, for example, uses about 5 000 blocks just to start up, and when loading the command line history etc., its size can easily be doubled.



Picture 1. Virtual address space of an RZMA process.

The virtual address space of the RZMA task is shown in picture 1. The addresses shown there are for typical mach task, that is run under unix emulation. For clarity only two allocated blocks are shown. For real applications there usually are much more data areas, and their size could be anything from 1 page to size of the virtual memory.

3.0 Implementation of the Redzal (Red Zone Allocator)

3.1 Initialisation

The Redzal can be configured by `xmalloc_config()` function. This function allows a programmer to set the sizes of the red zone, empty zone and the reserved area. This function is also used to set the error detection mode of Redzal. In current implementation the Redzal can be in two different mode, `XMALLOC_CHECK_BOTTOM` and `XMALLOC_CHECK_TOP`. In future we are implementing two more modes. More about the modes of the Redzal in chapter 6.1.

Error detection mode can only be set before any calls to xmalloc function. When xmalloc is called first time it will initialise itself using current mode, after that the mode cannot be changed anymore.

In the initialisation the Redzal will then try to locate the largest contiguous area of available virtual memory. Redzal will leave some empty space called Reserved area in the beginnign of this located space, and after the reserved area it will allocate enough space for its own tables. The rest of the area is used for the allocated blocks.

3.2 Exceptions

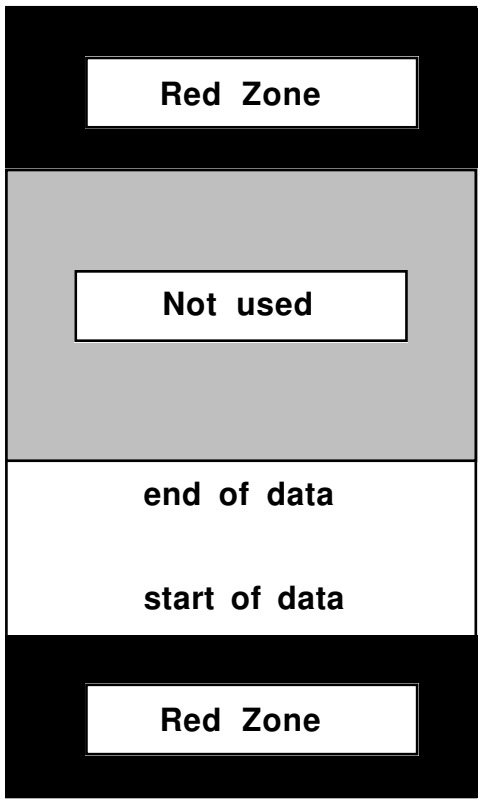
If the `xmalloc_debugger` variable is false, will create a separate exception thread for the current thread. Otherwise the debugger is supposed to handle exceptions. When an exception is raised, the exception thread calls `xmalloc_fault()` to print information about exception. If the exception was `EXC_BAD_ACCESS`, `xmalloc_fault()` routine will wait for 60 seconds to give the programmer time to start a debugger and attach to the task. After 60 seconds, the program will abort and dump core. If the exception is not `EXC_BAD_ACCESS` it is forwarded to the original exception handler (normally the unix server).

3.3 Allocation

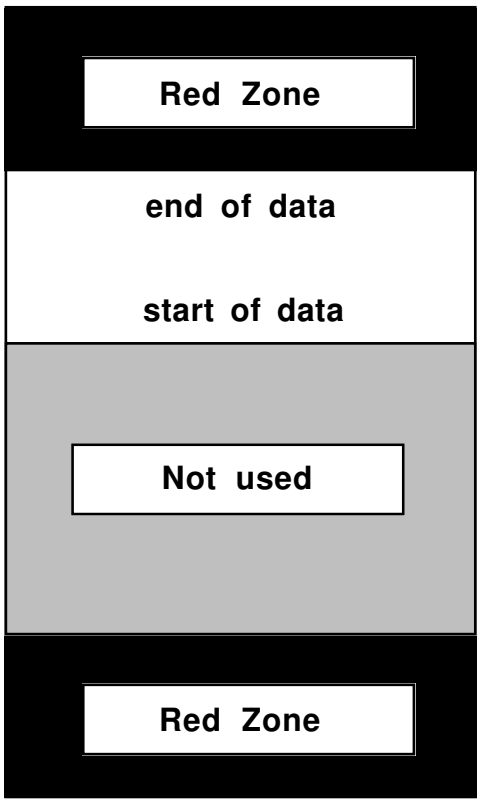
Due to limitations of the hardware virtual memory system in the conventional machines Redzal can only detect either top or

bottom violations of allocated block, not both. This is because the hardware will only allow permissions to be set on a per page basis and not per byte. Redzal has to allocate at least one page of memory with read and write permissions for the data, and another pages around the data pages, for the red zones with no access permission. Because users can allocate blocks of any size Redzal cannot check both the top and bottom limits of the blocks simultaneously. If it is configured to detect reads and writes before the start of the block, it will align the block so that it will start from the beginning of the page (see picture 2). This mode is called `XMALLOC_CHECK_BOTTOM`. When checking the end of the block, it will align the block so that it will end at the page boundary (see picture 3). This mode is called `XMALLOC_CHECK_TOP`.

Redzal can currently only be configured to one of these two modes. The default value is to select randomly either `XMALLOC_CHECK_TOP` or `XMALLOC_CHECK_BOTTOM`.



Picture 2. The allocated block when checking the bottom



Picture 3. The allocated block when checking the top

3.4 Freeing data

When data is freed the pages used for it are deallocated, and the memory is immediately returned to the operating system removing it from the address space of the user program. Information about the freed block will still be in the data block table, and the virtual memory area used by the block is left reserved. Redzal will store the address of the call to `free()`, so it can print it if the program tries to access a freed block.

3.5 Reallocating data

Redzal will always allocate a new block for reallocated data and move the data there. After that, the old block is freed, so Redzal will detect all accesses to the old location of data.

3.6 Illegal access

When Redzal detects an illegal operation it will display the following information:

```
xmalloc: Accessed freed data
xmalloc: code = 1 (1) [Invalid access]
xmalloc: Information about accessed data:
xmalloc: Object start address = 0x402000 (4202496)
xmalloc: Object size          = 0x202 (514)
xmalloc: Mallocated from      = 0x10040 (-)
xmalloc: Freed from           = 0x1044f (-)
xmalloc: Accessed address     = 0x402002 (4202498)
xmalloc: Called from          = 0x10899 (-)
You have 60 seconds time to attach this process before
abort
```

This contains enough information for the programmer to find the bug and fix it. If more information is needed the programmer can use a debugger and attach to the task.

4.0 Special features

4.1 Changing the protection of block

The programmer can change the protection of any allocated block with `xmalloc_setopt()`. Available permissions are `XMALLOC_PERM_RW` (the default value), `XMALLOC_PERM_RO` (for read only data) and `XMALLOC_PERM_NA` (no access). `xmalloc_getopt()` can be used to test permissions of a block. This feature can be used to detect illegal modifications or reading of data in the following way.

If the programmer knows that somewhere in the program some data is overwritten, the programmer can remove write permissions from the overwritten block and allow writing to the block only in the proper places. If the program tries to modify the block outside those explicit places this will be detected.

4.2 Permanent and temporary blocks

When a block is created it is marked as temporary by default. The programmer can change the state of the block to permanent by xmalloc_setopt using the flavour `XMALLOC_PERMANENT`, and back to temporary by the flavour `XMALLOC_TEMPORARY`. If a block is permanent it is illegal to free it. With `xmalloc_dump()` the programmer can print information about all temporary blocks or about all blocks. This feature can be used to find memory leaks.

4.3 Malloc breakpoint

Redzal has one special user settable breakpoint (`xmalloc_break_if_touched`) that can be set to contain start of any allocated block. When Redzal does any operations (malloc, free, realloc, etc.) involving this block, it calls the function `xmalloc_break()`. This function will print an informative message telling what the program tried to do and then continue normally. By setting normal a breakpoint in this function the programmer can break every time the program does something to a specific block. To use `xmalloc_break_if_touched` the programmer runs the program normally until he can find the address of the offending block (either by knowing it beforehand from previous executions or by detecting the allocation of the block), then by setting the offending address to the `xmalloc_break_if_touched` variable and setting breakpoint to `xmalloc_break()`. Redzal will break first when block is allocated, next when it is freed, resized etc.

5.0 Compatibility

By default Redzal is not fully compatible with normal malloc. It does not align blocks correctly to a double word boundary when checking top, because it aligns blocks strictly to the top of a page in this mode. This usually does not cause problems, because normally the size of the block is aligned properly for the data in it. However, in some cases programmers combine different kinds of data (e.g. a structure and a string) and that will cause trouble if the machine needs proper alignment. This can be avoided by setting the variable `xmalloc_align` to true. Redzal will then align all blocks correctly, although it cannot then detect all errors when

checking the top. This is because there may be a small gap between the end of the data and the red zone.

When allocating 0 bytes of memory, Redzal will print a warning and allocate only the red zones. If program tries to access this block, Redzal will trap this.

Old use of `free()`, `realloc()`, `free()`, `realloc()` combination to pack data is not supported, because this feature should not be used. Read the manual page of the malloc for more information about this "feature".

6.0 Interface functions

6.1 xmalloc_config

```
xmalloc_err_t xmalloc_config(xmalloc_check_t type,
          vm_size_t red_zone_size,
          vm_size_t empty_zone_size,
          vm_size_t reserved_zone_size)
```

Configure Redzal. Type can be only set before any calls to malloc. Legal values are:

`XMALLOC_CHECK_BOTTOM`: Will catch all accesses below an allocated block.

`XMALLOC_CHECK_TOP`: Will catch all access beyond the end of a block. If `xmalloc_align` is true, Redzal cannot catch all errors, but will be compatible with standard malloc.

`XMALLOC_CHECK_BOTH_READ`, `XMALLOC_CHECK_BOTH_WRITE`: Not yet implemented. Will be implemented by disallowing access to the data, trapping all accesses or only writes to the block, and single stepping over the offending instruction.

The last two settings are not real modes, but only affects how the real mode is selected.

`XMALLOC_CHECK_RANDOM`: Randomly select either check bottom or top. It will print which one it selected to stderr.

`XMALLOC_CHECK_ENVIRONORRAND`: Either use values in the `XMALLOC` environment variable or same as `XMALLOC_CHECK_RANDOM`. The `XMALLOC` environment variable has following syntax:

"<type>,<red>,<empty>,<reserved>,<alignment>"

Any of the fields can be empty. <Type> is a string setting Redzal mode. Possible values are: BR = Both read, BW = Both write, B = Bottom, T = Top and R = Random. <Red>, <empty> and <reserved> are numbers telling sizes of red, empty and reserved zone. Those numbers can have either 'k' or 'M' after them meaning 'kilobyte' or 'megabyte'. The <alignment> field can have the value 'A' if strict alignment is needed. Default value is "R,4k,4k,8M,"

The red_zone, empty_zone and reserved_zone arguments will set the sizes of the red, empty and reserved zone respectively. The environment always overrides the settings with this call if type is set to XMALLOC_CHECK_ENVIRONORRAND.

6.2 libmalloc: malloc, free, calloc, realloc, strdup

```
void *malloc(vm_size_t size)
void free(void *p)
void *calloc(vm_size_t n, vm_size_t size)
void *realloc(void *p, vm_size_t size)
char *strdup(const char *s)
```

These functions are the replacements for those in libc.

6.3 libxmalloc: xmalloc, xfree, xcalloc, xrealloc, xstrdup

Because many programs already have some kind of checked malloc or using the default names might cause trouble, the names of the replacement functions are changed so that all they have 'x' prefix. If program has its own xmalloc functions they must be removed and replaced with those in this library, because it is no use to know that this block was allocated from the function called xmalloc, which would be the case if you only linked program with libmalloc. The another way to compile program is by setting CFLAGS = -Dmalloc=xmalloc -Dfree=xfree -Dcalloc=xcalloc -Drealloc=xrealloc, and linking it with this library.

6.4 libymalloc: ymalloc, yfree, ycalloc, yrealloc, ystrdup

Same as libxmalloc, but 'y' prefix.

6.5 xmalloc_setopt and xmalloc_getopt

```
void xmalloc_setopt(void *p,
          xmalloc_operation_t operation)
boolean_t xmalloc_getopt(void *p,
          xmalloc_operation_t operation)
```

Set and get options for the blocks. Legal values for operation are:

`XMALLOC_PERM_RW`: Set the permissions to read / write, or return true if the block has read / write permissions.
`XMALLOC_PERM_RO`: Set the permissions to read only, or return true if the block has read only permission.
`XMALLOC_PERM_NA`: Remove all permissions from the block, or return true if the block has no permissions.
`XMALLOC_PERMANENT`: Mark the block as permanent, or return true if the block is permanent.
`XMALLOC_TEMPORARY`: Mark the block as temporary, or return true if the block is temporary.

6.6 xmalloc_dump

```
void xmalloc_dump(boolean_t all)
```

Dump information about blocks. If <all> is true then print all blocks, otherwise print only temporary blocks.

6.7 xmalloc_break

This function is only for setting breakpoint to it. It is called when `xmalloc_break_if_touched` variable matches the current block address.

7.0 Interface variables

7.1 xmalloc_align

```
boolean_t xmalloc_align = FALSE
```

If true, then all blocks are aligned properly even if checking top.

7.2 xmalloc_debugger

```
boolean_t xmalloc_debugger = FALSE
```

If true, no exception handler is installed. Used previously under debugger, but now gdb can handle multiple threads correctly, so this variable has no use anymore.

7.3 xmalloc_break_if_touched

```
vm_address_t xmalloc_break_if_touched
```

If this variable is set then all addresses passed to RZMA will be checked against this. Function `xmalloc_break()` will be called if match is found.

8.0 Future enhancements

The Redzal could be more efficient if it had its own external memory manager. With this it could lower its memory usage to a more user-friendly level. An EMM could cache a fixed amount of pages for allocated data, and pack all other data to its own memory. From there the default pager can page them to disk efficiently. Because most of the allocated blocks are much smaller than one page, memory requirements would be about 100 times smaller than now.

Another feature that has been planned is to implement checking of both ends of block in one run by protecting all blocks, and enabling permissions only during single stepping over offending instructions. This would slow down execution of the program, but it also offers other new features like storing last usage of a block (address and time), so memory leaks can be more easily detected.

RZMA could also detect that the first access to a block must be a write. With this it could detect programs that assume that malloc will return zeroed data.